



**UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA**

GRADO EN INGENIERÍA EN INFORMÁTICA

**Gemelo digital de un motor de inducción con
sensor de vibración para aplicaciones de
mantenimiento industrial**

Daniel Espinar Jiménez

Julio, 2022

**GEMELO DIGITAL DE UN MOTOR DE INDUCCIÓN CON SENSOR DE
VIBRACIÓN PARA APLICACIONES DE MANTENIMIENTO INDUSTRIAL**



**UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA**

Tecnologías y Sistemas de Información

**GRADO EN INGENIERÍA EN INFORMÁTICA
INGENIERÍA DE COMPUTADORES**

**Gemelo digital de un motor de inducción con
sensor de vibración para aplicaciones de
mantenimiento industrial**

Autor: Daniel Espinar Jiménez

Tutor académico: Félix Jesús Villanueva Molina

Julio, 2022

Daniel Espinar Jiménez

Ciudad Real – España

© 2022 Daniel Espinar Jiménez

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Se permite la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de Documentación Libre GNU, versión 1.3 o cualquier versión posterior publicada por la *Free Software Foundation*; sin secciones invariantes. Una copia de esta licencia esta incluida en el apéndice titulado «GNU Free Documentation License».

Muchos de los nombres usados por las compañías para diferenciar sus productos y servicios son reclamados como marcas registradas. Allí donde estos nombres aparezcan en este documento, y cuando el autor haya sido informado de esas marcas registradas, los nombres estarán escritos en mayúsculas o como nombres propios.

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE

VOCAL

SECRETARIO

Fdo.:

Fdo.:

Fdo.:

Resumen

En el presente documento se explicará proceso de diseño, desarrollo e implementación de un gemelo digital. La intención es que este documento sirva primero para comprender el propósito de los gemelos digitales en la industria, así como las ventajas que estos pueden ofrecer, y segundo, para poder seguir el proceso de diseño y desarrollo no solo de un único gemelo digital, sino de un sistema que permita el manejo de varios gemelos digitales.

Este documento aprovecha para proporcionar información sobre el origen de la idea del gemelo digital, así como las principales características de estos. Para mostrar el proceso de desarrollo se explicará el procedimiento lógico seguido detrás de las diferentes decisiones tomadas. De esta forma se podrá tener el suficiente conocimiento sobre el sistema para ser capaz realizar modificaciones o añadir funciones de una forma sencilla con el objetivo de que se adecúe a las necesidades de un proyecto concreto.

Al final del proceso se tendrá el diseño de un sistema escalable de gemelos digitales junto con la implementación de un prototipo de gemelo digital utilizando equipo real para demostrar el funcionamiento.

Abstract

This document will explain the process of designing, developing and implementing a digital twin. The intention is that this document serves first to understand the purpose of digital twins in the industry as well as the advantages that these can offer, and second, to be able to follow the process of design and development of not only a single digital twin, but of a system that allows the management of several digital twins.

This document provides information on the origin of the idea of the digital twin as well as the main characteristics of these. To show the development process, the logical procedure followed behind the different decisions taken will be explained. This allows to have enough knowledge about the system to be able to make modifications or add functions in a simple way so that it fits the needs of a specific project.

At the end of the process there will be the design of a scalable system of digital twins along with the implementation of a prototype of digital twin using real equipment to demonstrate the performance.

Agradecimientos

Quisiera aprovechar este espacio para agradecer a las personas que me han acompañado durante esta etapa de la vida. En primer lugar, a mis familiares, por todo lo que me han apoyado y facilitado lo que fuera necesario durante todo el proceso. También a aquellos compañeros con los que he compartido esta etapa y se han convertido en amigos. Y por último al tutor de este trabajo y en general a aquellos profesores que realizan su trabajo de forma adecuada y ayudando a los alumnos cuando estos lo necesitan. Gracias.

Daniel E.J.

*A mis familiares
Por apoyarme durante toda esta etapa*

Índice general

Resumen	V
Abstract	VII
Agradecimientos	IX
Índice general	XIII
Índice de cuadros	XVII
Índice de figuras	XIX
Índice de listados	XXI
Listado de acrónimos	XXIII
1. Introducción	1
1.1. ¿Por qué usar un gemelo digital?	1
1.1.1. Monitoreo remoto	1
1.1.2. Historial	2
1.1.3. Estrategias de mantenimiento	2
1.1.4. Simulación de escenarios	2
1.2. ¿Cómo funciona?	3
1.3. Prototipo	3
1.4. Estructura del documento	3
2. Objetivos	5
2.1. Objetivo general	5
2.2. Objetivos específicos	5
2.2.1. Objetivo específico 1	5
2.2.2. Objetivo específico 2	5

0. ÍNDICE GENERAL

2.2.3. Objetivo específico 3	6
2.2.4. Objetivo específico 4	6
3. Metodología	7
3.1. Proceso de desarrollo de software	7
3.2. Metodología de desarrollo software	8
3.2.1. Desarrollo Rápido de Aplicaciones (RAD, <i>Rapid Application Development</i>)	9
3.3. Proceso de testing	11
3.4. Herramientas utilizadas	11
3.4.1. Motor gráfico	12
3.4.2. IDE (Integrated Development Environment)	12
3.4.3. Lenguaje de programación	12
3.4.4. Middleware de comunicaciones	13
3.4.5. Gestor de base de datos	13
3.4.6. Depuración	14
3.4.7. Testing	14
3.4.8. Modelado 3D	15
3.4.9. Repositorios y control de versiones	15
3.4.10. Métricas y medidas	15
3.4.11. Documentación	16
3.5. Equipo físico	16
3.5.1. Servidor	16
3.5.2. Cliente del entorno 3D	16
3.5.3. Cliente lector del sensor	17
4. Resultados	19
4.1. Diseño	19
4.1.1. Diseño de la arquitectura	19
4.1.2. Diseño del modelo 3D	23
4.1.3. Diseño de la Base de datos	25
4.1.4. Diseño de la interfaz <i>ZeroC Ice</i>	27
4.2. Implementación	31
4.2.1. Implementación del sub-objetivo 1	31
4.2.2. Implementación del sub-objeteivo 2	38
4.2.3. Implementación del sub-objetivo 3	44

4.2.4. Script de actualización de los límites	47
4.2.5. Implementación del sub-objetivo 4	48
4.3. Pruebas	53
4.3.1. Pruebas del servidor	53
4.3.2. Pruebas del cliente del entorno 3D	55
4.3.3. Pruebas del cliente lector de sensores	56
4.4. Costes, planificación y presupuesto	56
5. Conclusiones	59
5.1. Trabajos futuros	60
5.2. Justificación de competencias adquiridas	61
A. Repositorio	65
Referencias	67

Índice de cuadros

4.1. Categorías de datos	28
4.2. Valores obtenidos del funcionamiento normal del motor eléctrico	53
4.3. Límites establecidos para detección del correcto funcionamiento	54
4.4. Costes de los recursos hardware	57
4.5. Costes de los recursos humanos	57

Índice de figuras

1.1. Arquitectura del prototipo final	3
1.2. Interfaz gráfica final	4
3.1. Modelo espiral de Boehm	9
3.2. Fases de la metodología	10
3.3. Motor eléctrico	17
3.4. Sensor CISS	18
3.5. Raspberry Pi	18
4.1. Diseño de arquitectura mínima	20
4.2. Diseño de arquitectura básica	20
4.3. Diseño de arquitectura algo escalable	21
4.4. Diseño de arquitectura escalable	21
4.5. Diseño de arquitectura final	22
4.6. Ejemplo de rotoscopia	23
4.7. Planos del motor	24
4.8. Modelo 3D del motor eléctrico	25
4.9. Tabla de lecturas	26
4.10. Tablas de identificación en la base de datos	27
4.11. Diseño final de la base de datos	28
4.12. <i>Script</i> de conexión	40
4.13. <i>Scripts</i> de la gráfica de valores	40
4.14. Componente gráfica de valores	41
4.15. Objeto de representación de datos de tres valores	41
4.16. Objeto de representación de datos de un valor	42
4.17. Ventana de información del gemelo digital	42
4.18. Interfaz del usuario	43
4.19. Ventana de información en 3D	43
4.20. Valores simulados	44

0. ÍNDICE DE FIGURAS

4.21. Motor eléctrico del prototipo	49
4.22. Sensor acoplado al motor	49
4.23. Modelo 3D integrado en el entorno 3D	51
4.24. Especificación de los identificadores	52
4.25. Valores generados con cliente de pruebas	56

Índice de listados

4.1.	Definición de la estructura <i>DataSet</i>	28
4.2.	Funciones <i>put</i> y <i>get</i>	29
4.3.	Definición de las estructuras <i>SingleDataLimits</i> y <i>DataSetLimits</i>	29
4.4.	Funciones <i>updateLimits</i> y <i>getLimits</i>	30
4.5.	Archivo de configuración del servidor	31
4.6.	Inicio de sesión en la base de datos	32
4.7.	Definición de las tablas con <i>sqlalchemy</i>	32
4.8.	Implementación de las funciones <i>get</i>	33
4.9.	Implementación de las funciones <i>put</i>	34
4.10.	Implementación de las funciones <i>updateLimits</i>	35
4.11.	Inicialización del entorno <i>ZeroC Ice</i> del servidor	36
4.12.	Archivo Dockerfile del servidor	37
4.13.	Inicialización del entorno <i>ZeroC Ice</i> del cliente de representación 3D	39
4.14.	Archivo de configuración del cliente que lee los sensores	45
4.15.	Inicialización del entorno <i>ZeroC Ice</i> del cliente lector de sensores	45
4.16.	Envío de datos al servidor	46
4.17.	Archivo de configuración de los límites	47
4.18.	Parámetros de configuración extra	50
4.19.	Envío de datos del sensor al servidor	50
4.20.	Pruebas unitarias al servidor	54
4.21.	Configuración del nodo cliente de pruebas	55

Listado de acrónimos

SDLC	Software Development Life-Cycle
SSADM	Structured Systems Analysis and Design Methodology
OOD	Object-Oriented Design
RAD	Rapid Application Developmnet
OOP	Object Oriented Programming
CISS	Connected Industrial Sensor Solution
PP	Point to point
OOP	Object-Oriented Programming

Capítulo 1

Introducción

Un gemelo digital es la representación virtual de un producto, proceso o servicio, que refleja en todo momento el estado y/o funcionamiento de su contraparte real. Un gemelo digital puede utilizarse para evaluar el funcionamiento, su estado y la información relacionada de la contraparte real, además de poder utilizarse para predecir comportamientos a futuro dependiendo de la implementación.[12]

En este caso se tomarán en cuenta los datos proporcionados por un sensor acoplado al motor real siendo este un caso básico. En desarrollos a mayor escala con más recursos pueden utilizarse incluso varios gemelos digitales para monitorear una sola máquina, correspondiendo cada gemelo a una parte específica de la máquina. También pueden incluirse actuadores para que se realice alguna función si fuese necesario, como por ejemplo apagar un motor para evitar daños.

1.1 ¿Por qué usar un gemelo digital?

Como se ha mencionado previamente, los gemelos digitales pueden usarse para una gran variedad de tareas entre las que están la detección de anomalías, manejo de recursos, gestión de sistemas, análisis del comportamiento de dicho sistema, simulación...

Realizar estas tareas, sin la capacidad de simular y estudiar el comportamiento previamente, puede ser muy costoso tanto en recursos como en tiempo, ya que aunque existen herramientas de simulación, en algunos proyectos pueden no existir herramientas que simulen los aspectos necesarios.

Es por esto que el uso de gemelos digitales trae muchos beneficios, ya sea para monitorear sistemas existentes como para diseñar sistemas nuevos.

1.1.1 Monitoreo remoto

Como ya se ha mencionado previamente una ventaja de utilizar gemelos digitales es la capacidad de monitorear el funcionamiento de motores reales de manera remota.

Por ejemplo, en una fábrica puede haber cientos de elementos que podrían fallar (máquinas, motores, puertas...) y no siempre se detecta el malfuncionamiento a simple vista, o incluso, no siempre hay alguien presente que pueda identificar el malfuncionamiento. Es por

1. INTRODUCCIÓN

esto que resulta extremadamente útil tener la posibilidad construir un entorno 3D que muestra todo lo que está pasando. Además el monitoreo remoto permite *visualizar* y *controlar* todos los elementos de la fábrica desde un único punto, recibiendo *alertas* en caso de ser necesario y teniendo la capacidad de actuar para evitar problemas y/o daños, por ejemplo deteniendo una máquina o apagando un motor.

1.1.2 Historial

Una función importante de los gemelos digitales es que permiten almacenar un historial de los estados de la contraparte real. El modelo digital representa en todo momento el estado de la parte real, por lo que a medida que el tiempo avanza, los estados pasados son almacenados convirtiéndose en el historial de la parte real.

Evidentemente el tipo de información que se guardará dependerá del objetivo que tenga el gemelo digital, evitando información innecesaria y centrándose en aquella de interés para el usuario. Por ejemplo, si el objetivo del modelo digital está ligado al análisis del rendimiento bajo ciertas circunstancias será más interesante guardar datos que puedan ser comparables para realizar comparaciones con otros modelos en las mismas circunstancias o con el mismo modelo en condiciones diferentes.

El historial también es útil para el *machine learning*, ya que permitiría clasificar anomalías, y para análisis forense en caso de que fuese necesario, ya que permite analizar que puede haber pasado en un momento concreto.

1.1.3 Estrategias de mantenimiento

La capacidad de monitorear un sistema de manera centralizada, también da la capacidad de monitorear como se comportará este en caso de que algún componente falle. Esto es extremadamente útil sobre todo si se esperan fallos de algún componente en el futuro, deshabilitar algún nodo para mantenimiento, etc...

Gracias a esto se pueden determinar nuevas estrategias de mantenimiento para que se adecuen de la mejor manera posible al sistema que se gestiona, reduciendo así los costes de los fallos o mantenimiento.

1.1.4 Simulación de escenarios

Al igual que un modelo digital puede ayudar a la hora de analizar estados pasados y presentes de algo real, también pueden ser útiles para *simular escenarios futuros*, de tal forma que se puede comprobar cómo le afectarían los diferentes factores como pueden ser el entorno, el tamaño de un sistema u otras condiciones que puedan afectar al funcionamiento, y por tanto, al rendimiento.

1.2 ¿Cómo funciona?

La idea detrás del funcionamiento de un gemelo digital es muy sencilla. Por ejemplo, para crear una representación actualizada de un motor, simplemente hay que construir un modelo 3D de dicho motor y hacer que este vaya actualizándose según los valores que recibe. Estos valores pueden ser datos leídos por los sensores de un motor real, lo que nos serviría para realizar un monitoreo, valores simulados, lo que nos serviría para realizar una simulación, o datos obtenidos mediante algunas operaciones con otros valores existentes, lo que nos serviría para mantenimiento predictivo, por ejemplo.

De esta forma, teniendo varios gemelos digitales, se puede realizar un seguimiento de toda la flota de motores de una compañía desde un punto central, realizar pruebas añadiendo y/o eliminando motores para comprobar el funcionamiento global del sistema, analizar el sistema, etc...

1.3 Prototipo

Al final del presente documento se habrá explicado el procedimiento y razonamientos seguidos hasta llegar al diseño un sistema y el desarrollo de un *prototipo*.

La arquitectura de un sistema amplio no coincide con la de un prototipo y en este caso la arquitectura final del prototipo que se construirá es la mostrada en la figura 1.1.

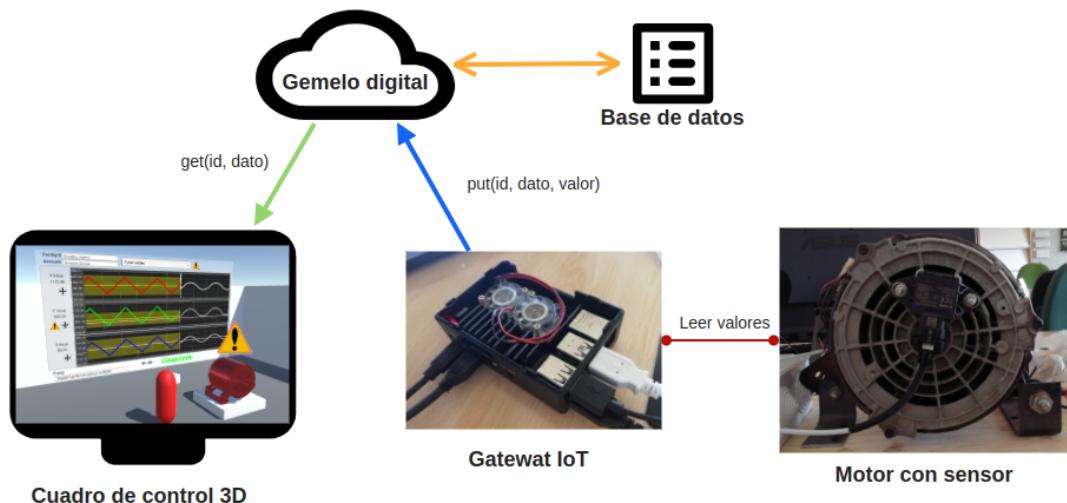


Figura 1.1: Arquitectura del prototipo final (Figura del autor)

Por la parte visual, el resultado final de la interfaz del entorno 3D puede verse en la figura 1.2.

1.4 Estructura del documento

En esta sección se explicará brevemente la estructura de este documento y sus contenidos. En los capítulos siguientes se describirá el proceso de diseño y desarrollo de un gemelo

1. INTRODUCCIÓN

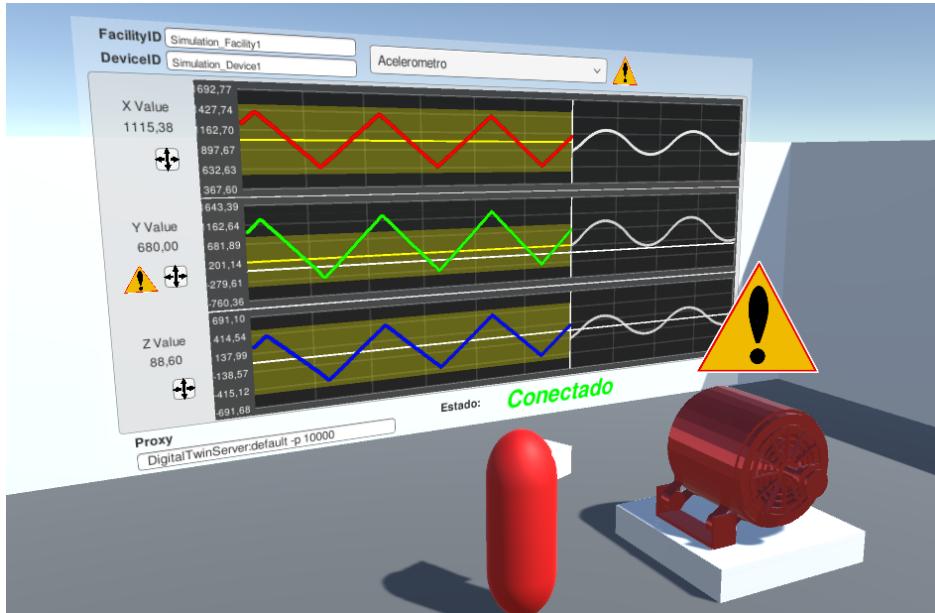


Figura 1.2: Interfaz gráfica final del presente proyecto (Figura del autor)

digital de un motor eléctrico explicando los diferentes sub-objetivos para alcanzar el objetivo general.

Capítulo 2: Objetivos

Finalidad y justificación del proyecto y el presente documento.

Capítulo 3: Metodología

Metodologías empleadas para planificación y desarrollo del presente trabajo, así como la explicación de cómo se han aplicado dichas metodologías.

Capítulo 4: Resultados

Aplicación del método de trabajo presentado en el capítulo 3, mostrando los elementos (modelos, diagramas, especificaciones, etc.) más importantes.

Capítulo 5: Conclusiones

Juicio crítico y discusión sobre los resultados obtenidos con la realización del presente trabajo.

Capítulo 2

Objetivos

2.1 Objetivo general

El principal objetivo del presente trabajo consiste en diseñar, implementar una arquitectura de gemelo digital para motores eléctricos con un entorno de pruebas que utilice un pequeño prototipo.

Este prototipo deberá mostrar en un entorno 3D los datos recopilados de un sensor acoplado a un motor eléctrico real.

2.2 Objetivos específicos

Este objetivo general se divide en varios sub-objetivos (u objetivos específicos) que permiten abordar el objetivo general a través de objetivos más pequeños y manejables. Nótese que los sub-objetivos no definen un orden de realización de estos ya que se pueden realizar de manera independiente.

2.2.1 Objetivo específico 1

El primer sub-objetivo consiste en diseñar y desarrollar un software que *recopile* y *almacene* los datos y el estado de determinados recursos físicos (ej. motor eléctrico) además de *proveer* esta información cuando se solicite.

Para el desarrollo de este software servidor se utilizará:

- El lenguaje de programación *Python* (versión *Python 3*)
- El middleware *ZeroZ Ice*
- El administrador de bases de datos *Postgresql*

2.2.2 Objetivo específico 2

El segundo sub-objetivo consiste en diseñar y desarrollar un software que muestre una interfaz de visualización 3D con capacidad de predicción/simulación y representación de la información asociada a un recurso físico.

En este caso se utilizará el software *Unity*, que utiliza el lenguaje de programación *C#*, para crear el entorno 3D. También será necesario el uso de *ZeroC Ice*.

2. OBJETIVOS

2.2.3 Objetivo específico 3

El tercer sub-objetivo consiste en el diseño y desarrollo de un software que obtenga datos de los recursos físicos (ej. motor eléctrico).

Para el desarrollo de este sub-objetivo se utilizará:

- El lenguaje de programación *Python* (versión *Python 3*)
- El middleware *ZeroZ Ice*

2.2.4 Objetivo específico 4

Otro sub-objetivo consiste en el diseño y desarrollo de un *entorno de pruebas* que muestre el correcto funcionamiento en conjunto de los componentes del sistema.

Para este entorno de pruebas se reutilizará un motor antiguo, el cual pertenecía a una lavadora, y se desarrollará un modelo 3D que represente dicho motor eléctrico. Para el desarrollo del prototipo se utilizará:

- Un modelo 3D diseñado utilizando el software *Blender*.
- Un sensor de la compañía BOSCH así como el código que esta proporciona¹ para el leer los datos de dicho sensor.
- Una *Raspberry Pi* como equipo que ejecutará el software y al cual se conectará el sensor

¹<https://www.bosch-connectivity.com/products/industry-4-0/connected-industrial-sensor-solution/downloads/>

Capítulo 3

Metodología

En este capítulo se detallarán las metodologías empleadas para planificación y desarrollo del trabajo, y se explicará de manera clara y concisa cómo se han aplicado dichas metodologías.

El presente proyecto consiste en diseñar y desarrollar un gemelo digital de un motor eléctrico. Ahora bien, como se ha mencionado anteriormente, hay que tener en cuenta todas las características del proyecto, así como los recursos disponibles para su realización (presupuesto, tiempo, equipo de trabajo...).

Es por esto que se diseñará la arquitectura de un sistema que pueda abarcar varios gemelos digitales y gestionarlos adecuadamente, sin embargo, se desarrollará únicamente un prototipo de gemelo digital que muestre el funcionamiento de un gemelo digital básico.

3.1 Proceso de desarrollo de software

Una metodología se puede definir como el modo en que se organizan las diferentes fases de un proyecto y como interaccionan para conseguir una productividad y calidad adecuadas. Estas fases se conocen comúnmente como *ciclo de vida* del desarrollo del software o *Software Development Life-Cycle (SDLC)* [9], y cubre las siguientes actividades:

1. *Planificación(Planning)*. En esta fase los líderes o encargados del proyecto evalúan los términos de este. Esto incluye calcular costes, marcar objetivos y crear la estructura y equipos de trabajo entre otras cosas.
2. *Obtención y análisis de requisitos(requirements analysis)*. En esta etapa se define el dominio del software, es decir, qué servicios debe proporcionar el sistema a desarrollar. En esta actividad se suele trabajar con los clientes y/o usuarios finales del sistema para poder obtener de forma precisa el 'qué' y el 'cómo' a fin de obtener una *especificación funcional* del sistema.
3. *Diseño (SW design)*. En esta etapa se utiliza la información recolectada en la anterior. Trata en definir el sistema, desde su arquitectura y sus componentes, hasta las interfaces y otras características.
4. *Implementación (SW construction and coding)*. Esta etapa consiste en la codificación

3. METODOLOGÍA

del software con unas herramientas y lenguajes de programación. Es la fase en la que realmente se produce el desarrollo del software.

5. *Pruebas (testing and verification)*. Esta etapa consiste en comprobar y verificar el correcto funcionamiento del sistema con el objetivo de detectar los fallos lo antes posible para obtener un software de calidad. Para esto se realizan pruebas de varios tipos, entre ellas pruebas de *caja negra* y *caja blanca*. Las pruebas de *caja negra* únicamente se comprueba que la salida es la esperada para una entrada determinada y las pruebas de *caja blanca* buscan verificar el correcto funcionamiento interno del software. También se realizan *pruebas de integración* en las que se comprueba el funcionamiento del sistema con otros existentes.
6. *Despliegue (deployment)*. En esta etapa se realiza la instalación del software en un entorno real y se pone en marcha para su uso. Según el tipo de software puede requerir una fase de *entrenamiento* de los usuarios.
7. *Mantenimiento (maintenance)*. Consiste en la detección y resolución de problemas del software, así como su mejora y adaptación a cambios en los requisitos.

3.2 Metodología de desarrollo software

Una metodología es una colección de:

1. *Procedimientos* (indican cómo hacer cada tarea y en qué momento)
2. *Herramientas* (ayudas para la realización de cada tarea)
3. *Ayudas documentales*

Existen diferentes tipos de metodologías siendo cada una apropiada para algunos tipos de proyectos u otros dependiendo de las características. Teniendo en cuenta la naturaleza del presente proyecto, así como sus características técnicas, recursos y equipo de trabajo, se elegirá una metodología que se adecúe de la mejor manera posible.

Algunas de las más conocidas son:

- Metodología de Análisis y Diseño de Sistemas Estructurados (SSADM, *Structured Systems Analysis and Design Methodology*) [1]
- Metodología de Diseño Orientado a Objetos (OOD, *Object-Oriented Design*) [18]
- Desarrollo Rápido de Aplicaciones (RAD, *Rapid Application Development*) [10]
- Metodologías Ágiles [22]

En este proyecto se utilizará la metodología de *Desarrollo Rápido de Aplicaciones* (RAD) ya que es la que mejor se adapta al presente proyecto.

3.2.1 Desarrollo Rápido de Aplicaciones (RAD, *Rapid Application Development*)

La filosofía de esta metodología consiste en sacrificar un poco la calidad a cambio de poner en producción un sistema con la funcionalidad esencial rápidamente. Los procesos de especificación, diseño e implementación, pertenecientes al ciclo de vida, se realizan de manera simultánea retroalimentándose unos a otros.

No se realiza una especificación detallada dado que constantemente se analizará el sistema y se propondrán cambios y nuevas mejoras según las necesidades y recursos del proyecto. Aunque pudiese parecer un modelo de desarrollo en cascada, realmente sigue un modelo de desarrollo en espiral (Ver Fig. 3.1), definido por primera vez por Barry Boehm cuya clave es el desarrollo continuo ayudando así a minimizar los riesgos.[13]

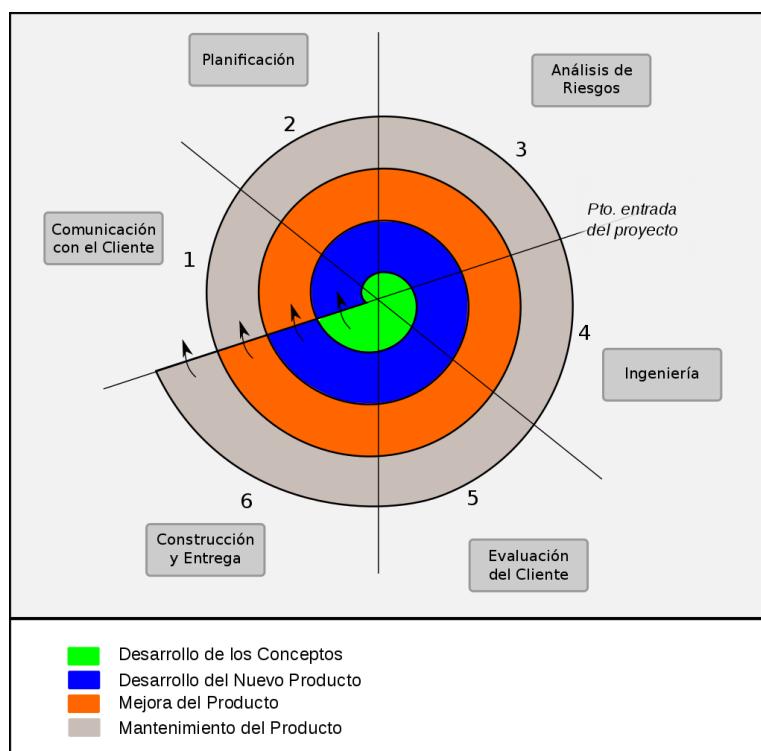


Figura 3.1: Figura del modelo espiral de Barry Boehm (por Ebnz en Wikipedia, CC BY-SA 3.0)

Este tipo de desarrollo se basa en la creación de prototipos y realimentación obtenida para definir e implementar nuevas características y/funcionalidades hasta alcanzar un sistema adecuado para el despliegue.

Como se puede ver esta metodología se adecúa perfectamente al presente proyecto por lo que será la que se utilice durante el ciclo de vida del proyecto. También cabe mencionar que, aunque es cierto que se utilizará una metodología global para el proyecto entero, se pueden apreciar algunas variaciones a la hora de trabajar en cada uno de los diferentes objetivos del sistema, ya que tiene necesidades distintas.

3. METODOLOGÍA

Para aplicarlo al presente proyecto se realizarán varias fases en las que se trabajará en todos los sub-objetivos de forma paralela empezando con diseños y/o funcionalidad básica los cuales se irán mejorando en las siguientes fases y se implementarán más funciones (ver fig.3.2).

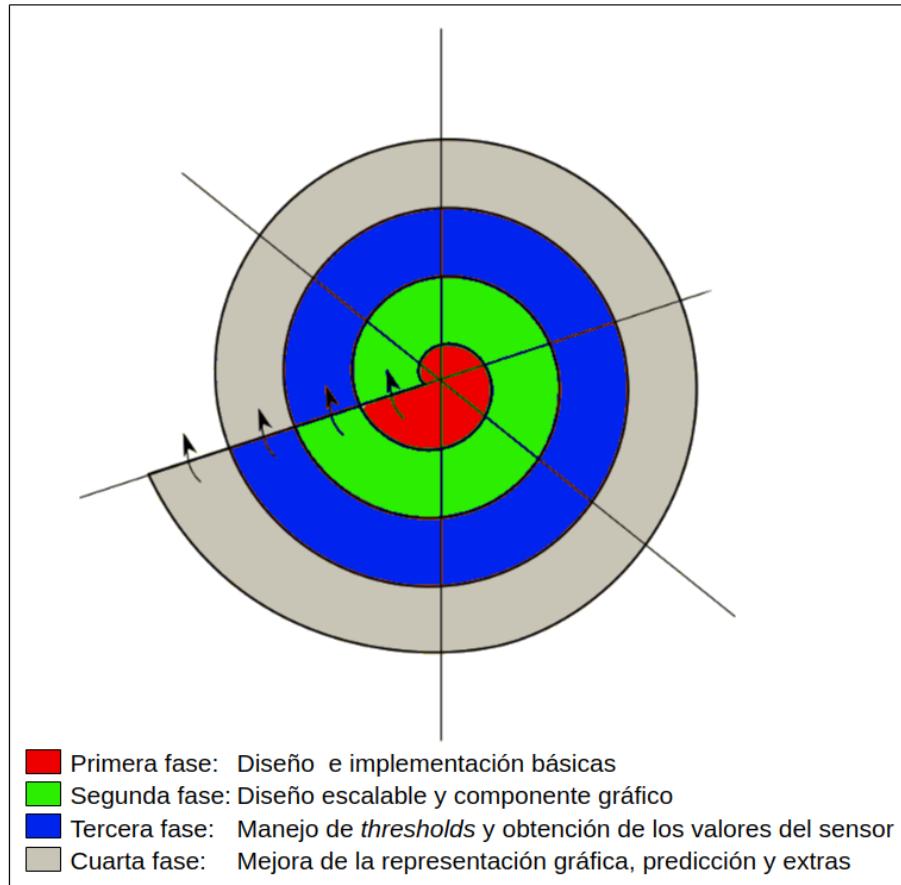


Figura 3.2: Fases de la metodología (Figura del autor)

1. En la primera fase se diseñará una arquitectura y una base de datos simples, y se implementarán los entornos *ZeroC Ice* con las funciones básicas *put* y *get*.
2. En la segunda fase se mejorarán los diseños de arquitectura y base de datos para permitir varios gemelos digitales y que por tanto sea un sistema escalable, adaptando convenientemente las implementaciones. Además se implementará una primera versión de un componente de representación gráfica en *Unity* y se acoplará el sensor al motor eléctrico real.
3. En la tercera fase se implementará lo necesario para manejar los *thresholds* (explicados posteriormente). Se adaptará la base de datos para poder manejar estos nuevos datos, además de adaptar el componente gráfico mencionado anteriormente para que represente dichos *thresholds*. En el servidor se implementarán las funciones necesarias para actualizar y obtener estos datos. Por otra parte, se implementará o reutilizará el código que lea los valores de los sensores y así poder enviarlos.

4. En la cuarta fase se implementarán las ventanas de información gráficas del entorno 3D y la predicción de valores futuros. Por otro lado, en el servidor se implementarán algunas funciones extra. También se creará un contenedor *docker* del servidor que permita su ejecución en cualquier máquina.

Tras esto se establecerá la configuración correcta en todos los componentes para permitir el correcto funcionamiento de un prototipo.

Cabe destacar que en cada fase se verificará el correcto funcionamiento de aquello que se implemente antes de pasar a la siguiente fase para evitar la acumulación de tareas y arrastrar errores. Además, durante todo el proceso se utilizarán archivos de configuración para facilitar modificaciones.

3.3 Proceso de testing

Para comprobar el correcto funcionamiento del gemelo digital es necesario realizar pruebas para todas las partes de este. Es altamente recomendable que las pruebas se vayan realizan a la par que se implementan o modifican las diferentes funcionalidades para poder detectar los fallos lo más rápido posible.

En este proyecto se realizarán varios tipos de pruebas diferentes:

1. *Pruebas de integración.* Pruebas de varios módulos en conjunto para comprobar su interoperabilidad.
2. *Pruebas de caja negra.* Son pruebas de software en las que se verifica el correcto funcionamiento teniendo en cuenta únicamente las entradas y las salidas, es decir, sin tomar en cuenta la estructura interna de código.

3.4 Herramientas utilizadas

En todo proyecto se utilizan diferentes herramientas que facilitan las diferentes tareas a lo largo del ciclo de vida del mismo. Algunas herramientas son de uso prácticamente obligatorio, como por ejemplo un motor gráfico, mientras que otras no son obligatorias, pero si altamente recomendables. En este caso se necesitarán herramientas de las siguientes categorías:

1. Motor gráfico
2. IDE (Integrated Development Environment)
3. Lenguaje de programación
4. Middleware de comunicaciones
5. Gestor de base de datos
6. Depuración
7. Testing
8. Modelado 3D

3. METODOLOGÍA

9. Repositorios y control de versiones
10. Métricas y medidas
11. Documentación

3.4.1 Motor gráfico

Una herramienta esencial a la hora de desarrollar un gemelo digital es el motor gráfico. Si bien es cierto que se podría crear un motor gráfico nuevo, ya existen varios que ofrecen muy buenas prestaciones y muchas posibilidades. Crear un motor gráfico nuevo no es una tarea sencilla y solo sería recomendable en caso de que ninguno de los existentes ofrezca las características y/o funcionalidades necesarias para el proyecto a desarrollar.

Para este proyecto se ha elegido el motor gráfico *Unity* [25]. Este motor es uno de los más conocidos y potentes en todo el mundo. Además, la página oficial de Unity ofrece una serie de cursos a modo de tutorial [24] que permiten obtener los conocimientos básicos para aprender a usar esta útil herramienta.

Algunos de las opciones a utilizar son:

- Unity
- Unreal Engine
- Godot

3.4.2 IDE (Integrated Development Environment)

También es altamente recomendable el uso de un *Entorno de Desarrollo Integrado*, o más comúnmente conocido como IDE (Integrated Development Environment). Estas herramientas hacen que la tarea del programador sea mucho más sencilla, ya que incluye otras herramientas como compiladores, depuradores, bibliotecas, etc..., lo cual se traduce en un aumento de la productividad.

Para este proyecto se ha elegido *Visual Studio Code* [27] ya que tiene paquetes que facilitan el desarrollo de código para la herramienta *Unity* [28], pero de nuevo existen multitud de opciones igualmente válidas:

- Visual Studio Code
- Notepad++
- GNU Emacs
- NetBeans

3.4.3 Lenguaje de programación

El lenguaje de programación se puede considerar una herramienta como tal, ya que todos son diferentes y unos ofrecen unas ventajas y desventajas frente a otros. Por ejemplo, mientras que unos lenguajes están diseñados para facilitar el OOP, otros están diseñados para facilitar la programación concurrente, etc... Es por esto que también hay que elegir adecua-

damente el lenguaje de programación si fuese posible, ya que algunas herramientas utilizan un lenguaje determinado y no puede cambiarse.

Cabe destacar que gracias a la modularización de un proyecto con varios módulos, se pueden utilizar diferentes lenguajes para cada módulo, utilizando en cada uno aquel que mejor se adecúe a las necesidades.

En este proyecto se ha elegido *Unity*, motor gráfico que utiliza el lenguaje *C#* y por tanto en el módulo/nodo que actuará como cliente ejecutando el entorno 3D no se podrá elegir el lenguaje de programación. Sin embargo, en el nodo que actúa como servidor y en el nodo que actúa como cliente leyendo los valores del sensor es posible elegir el lenguaje de programación que se quiera. En este caso se ha elegido el lenguaje *Python* [21].

Algunos de los lenguajes más conocidos son:

- Python
- Java
- C/C++
- C#

3.4.4 Middleware de comunicaciones

Para el presente proyecto será necesario el uso de algún middleware de comunicaciones, ya que cada uno de los nodos del sistema se ejecutará en equipos separados.

«Middleware es software que se sitúa entre un sistema operativo y las aplicaciones que se ejecutan en él. Básicamente, funciona como una capa de traducción oculta para permitir la comunicación y la administración de datos en aplicaciones distribuidas. ...El uso de middleware permite a los usuarios hacer solicitudes como el envío de formularios en un explorador web o permitir que un servidor web devuelva páginas web dinámicas en función del perfil de un usuario.»[14]

En el presente proyecto se utilizará el middleware *ZeroC Ice*. [29]

Usando el middleware *ZeroC Ice* simplemente habrá que definir una *interfaz* con las funciones y estructuras necesarias en el proyecto y posteriormente utilizar los *translators* específicos para cada lenguaje. De esta forma definiendo una única interfaz, *ZeroC Ice* generará automáticamente el código en el lenguaje que sea necesario. Una vez se tiene este código se podrá hacer uso de las estructuras de datos y funciones definidas previamente en la interfaz.

3.4.5 Gestor de base de datos

Para que el gemelo digital pueda proporcionar algunas de sus características, como puede ser el historial de estados, es necesario almacenar los datos en algún sitio. Es por eso que en el presente proyecto se hará uso de una base de datos que almacene toda la información

3. METODOLOGÍA

requerida, así como los valores de los sensores a medida que se leen.

Si bien podría implementarse un sistema propio para almacenar toda la información de diversas maneras, el uso de un gestor de bases de datos facilitará esta tarea. Además a este se le puede sumar el uso de un paquete que facilite la interacción con la base de datos permitiendo al desarrollador olvidarse de las sentencias SQL¹ ya que pueden llegar a ser demasiado complejas.

Existen varias opciones igualmente válidas para elegir y para este proyecto se utilizará el gestor *PostgreSQL* [17].

- PostgreSQL
- Amazon RDS
- MariaDB

3.4.6 Depuración

Otras herramientas muy útiles son las herramientas de depuración. Estas herramientas ayudan a la hora de detectar e identificar los problemas y su origen durante la fase de implementación, ahorrando mucho tiempo.

Existen varias herramientas de este tipo, de hecho el IDE utilizado en este proyecto (*Visual Studio Code*) incluye un módulo de depuración.

- GNU Debugger

3.4.7 Testing

También existen herramientas que facilitan el proceso de *testing* del proyecto, permitiendo generar pruebas para comprobar el correcto funcionamiento de nuestra implementación.

Según las pruebas necesarias para un proyecto será conveniente seleccionar una herramienta u otra. Para el presente proyecto, dado que se va a desarrollar un prototipo, será suficiente con realizar algunas pruebas de caja negra y finalmente pruebas sobre el sistema del prototipo en conjunto.

Dado que en este proyecto se ha elegido *Python* como lenguaje de programación para probar la parte del servidor se hará uso de *PyUnit* [20] para realizar algunas pruebas necesarias durante el desarrollo.

- PyUnit. Entorno de pruebas para Python.
- JUnit. Entorno de pruebas para Java.
- CUnit. Entorno de pruebas para C.

¹Las sentencias SQL son comandos SQL que se utilizan para interactuar con la base de datos

3.4.8 Modelado 3D

Dado que en un gemelo digital el motor eléctrico se representa en un entorno 3D, es conveniente que el modelo 3D sea lo más fiel posible al motor real. Para conseguir esto es conveniente hacer uso de alguna herramienta de modelado 3D que nos facilite esta tarea.

Hay una gran variedad de herramientas que nos pueden ayudar en esta tarea y para este proyecto se utilizará la herramienta de modelado 3D *Blender* [3]. Utilizando estas herramientas existen diversas técnicas diferentes para modelar un objeto.

Algunas herramientas de modelado 3D son:

- Blender
- Autodesk Maya
- Autodesk 3ds Max
- Rhinoceros

3.4.9 Repositorios y control de versiones

Los repositorios y/o herramientas de control de versiones son especialmente útiles para organizar el trabajo adecuadamente y permite que varias personas estén trabajando en partes distintas del proyecto simultáneamente. También permite realizar un seguimiento de los cambios que se han realizado, así como el contenido de estos, revisar los cambios antes de incluirlos definitivamente, asignar tareas...

De nuevo hay varias opciones para utilizar y se podrá elegir la que se prefiera. Para este proyecto se ha utilizado *Bitbucket* [2] y algunas opciones válidas son:

- Bitbucket
- Git
- Mercurial
- Github
- SourceTree

3.4.10 Métricas y medidas

Dependiendo del proyecto puede ser necesario alguna herramienta de métricas y medidas, ya sea para medir rendimientos, valores en determinadas situaciones, etc...

En cualquier sistema los nodos suelen tener una configuración inicial para poder realizar diferentes funciones y que puede ser modificada según las necesidades. En el presente proyecto, para poder diferenciar el funcionamiento apropiado de un funcionamiento inapropiado, será necesario medir el funcionamiento normal del motor real y posteriormente, haciendo uso de una herramienta de este tipo, calcular unos límites o *thresholds* para establecerlos en la configuración inicial mencionada previamente.

En el presente proyecto se hará uso de la herramienta *LibreOffice Calc* [11]. Algunas

3. METODOLOGÍA

opciones en esta categoría son:

- Microsoft Excel
- LibreOffice Calc
- Google Sheets

3.4.11 Documentación

La documentación es una parte esencial de todo proyecto por lo que es una buena idea hacer uso de herramientas que nos faciliten redactar una documentación con una presentación y estructura adecuadas y formales.

En la elaboración del presente documento se ha utilizado la herramienta L^AT_EX además de la plantilla ofrecida por parte de la Escuela Superior de Informática la cual está preparada para facilitar la aplicación de la guía de estilo [8].

Algunas herramientas conocidas son:

- L^AT_EX
- Markdown
- Doxygen
- DocGen
- Pandoc

3.5 Equipo físico

Como se ha mencionado anteriormente, en este proyecto se desarrollará un prototipo que muestre el funcionamiento de un gemelo digital de un motor eléctrico. Para llevar a cabo esta tarea será necesario disponer de equipo físico para poder implementar los diferentes componentes del sistema.

3.5.1 Servidor

Para la parte del servidor no sería necesario ningún equipo físico ya que se ejecutará en la nube, pero en caso de querer ejecutarlo en una máquina física propia, al ser un prototipo y no el sistema final, podría utilizarse cualquier equipo con antena Wifi o puerto ethernet para poder conectarse a Internet y capacidad de procesamiento y almacenamiento suficiente, ya que también almacenará la base de datos.

Para hacer pruebas con el prototipo debería ser suficiente con un ordenador portátil común.

3.5.2 Cliente del entorno 3D

Para el cliente que implementará el módulo de representación 3D no es necesario nada específico, tan solo un ordenador con capacidad de procesamiento suficiente para la carga de trabajo que supone el entorno 3D.

La capacidad necesaria dependerá de la cantidad de gemelos digitales que vayan a representarse en el entorno 3D ya que la carga aumentará a medida que aumente la cantidad de elementos en dicho entorno.

Para el prototipo solo se utilizará un gemelo digital por lo que la carga no será elevada.

3.5.3 Cliente lector del sensor

Para este nodo del proyecto si será necesario más equipamiento físico. Dado que en este proyecto se implementará el gemelo digital de un motor eléctrico, serán necesarios los siguientes elementos:

- *Motor eléctrico.* Motor real al cual se le colocarán los sensores/actuadores necesarios y del cual se realizará el seguimiento.
- *Sensores/Actuadores.* Sensores y/o actuadores que se acoplarán a la contraparte real del gemelo digital, en este caso, el motor eléctrico. Según la información que se quiera recolectar se podrán utilizar diferentes sensores.
- *Equipo procesador.* Equipo encargado de conectarse al servidor y mandarle los datos que extraerá de los sensores.

Motor eléctrico

Para desarrollar el prototipo de este proyecto se hará uso de un único motor eléctrico, más concretamente, se reutilizará un motor eléctrico viejo de una lavadora (Ver Fig. 3.3).

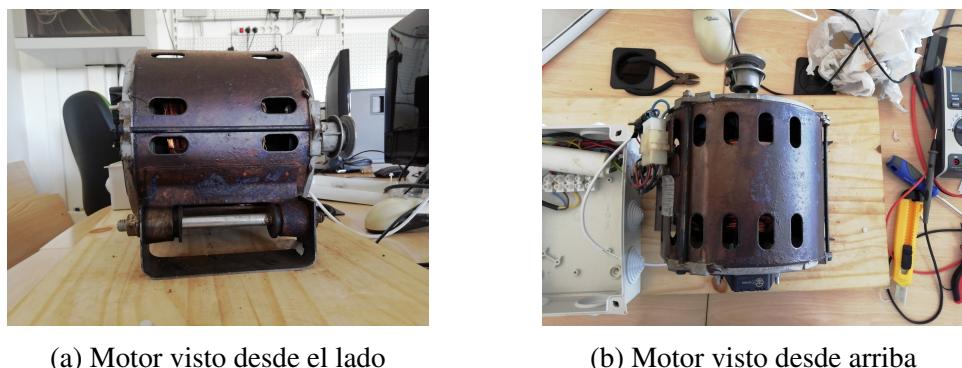


Figura 3.3: Motor eléctrico (Figura del autor)

Sensores y/o Actuadores

En este proyecto, el prototipo a desarrollar constará de un único sensor anclado al motor eléctrico. Dicho sensor será el sensor desarrollado por la empresa BOSCH denominado *Connected Industrial Sensor Solution (CISS)* (Ver Fig.3.4).

Este sensor ofrecerá información de:

- Acelerómetro (x, y, z)

3. METODOLOGÍA



Figura 3.4: Fotografía del sensor CISS (Fotografía de BOSCH)

- Temperatura
- Humedad
- Luz
- Giroscopio (x, y, z)
- Magnetómetro (x, y, z)
- Presión

Toda la información relativa al sensor, así como instrucciones de uso, pueden encontrarse en la página web oficial [4].

Equipo procesador

Por último, será necesario el equipo que se encargará de la lectura de los datos del sensor y de mandar dichos datos al nodo servidor. Dado que el sensor seleccionado se comunicará por interfaz USB este equipo deberá tener al menos un puerto USB.

En este caso dado que no se necesita una capacidad de procesamiento elevada se utilizará una Raspberry Pi (Ver Fig. 3.5).



Figura 3.5: Fotografía de la Rapberry Pi del prototipo (Fotografía tomada por el autor)

Capítulo 4

Resultados

En este capítulo se describirá la aplicación del método de trabajo presentado en el capítulo 3:Metodología, mostrando los elementos (modelos, diagramas, especificaciones, etc.) más relevantes del proceso.

Se explicará el proceso de diseño y de desarrollo realizado durante este proyecto justificando las diferentes decisiones tomadas para cumplir los diferentes objetivos específicos y finalmente el objetivo general.

4.1 Diseño

En esta sección se describirán los diseños realizados para el presente proyecto que consisten en las definiciones de arquitecturas, los diferentes componentes, como se interconectan, las interfaces y otras características del sistema o sus componentes.

Primero se explicará el diseño de la arquitectura general y posteriormente los diseños necesarios en diferentes nodos.

4.1.1 Diseño de la arquitectura

Primero se explicará el proceso de razonamiento que se ha seguido hasta llegar al diseño de arquitectura final. Se empezará con la arquitectura más básica posible y se irá mejorando hasta conseguir una arquitectura final que sea adecuada y escalable.

Diseño mínimo

En un primer vistazo un gemelo digital puede dividirse en dos componentes: el equipo donde se representará el entorno 3D y el motor real del que se leerá la información con los sensores (Ver Fig.4.1).

Sin embargo, esta arquitectura no incluye algunas características de un gemelo digital, como el monitoreo remoto, además de que es una solución que no aporta mucho al usuario. Si se implementase esta arquitectura el cliente visualizaría el entorno 3D en un equipo conectado al motor real por lo que podría ver también dicho motor. Esto solo sería útil en caso de que solo se quiera monitorear el funcionamiento de un único elemento (motor, máquina, puertas...) y sea de interés visualizar gráficamente los valores exactos que se producen.

4. RESULTADOS



Figura 4.1: Diseño de arquitectura mínima para un gemelo digital (Figura del autor)

Esta solución no sería escalable y tendría un alto coste en proporción con los beneficios.

Diseño básico

La característica más esencial de un gemelo digital podría considerarse que es el monitoreo remoto, por lo que ahora se modificará el diseño anterior (Ver Fig.4.1) para disponer de esta característica.

Para lograr este objetivo basta con separar la parte de representación del entorno 3D de la parte de lectura del sensor. Para esto se puede añadir un nuevo nodo al sistema resultando así en dos nodos diferenciados (Ver Fig.4.2).

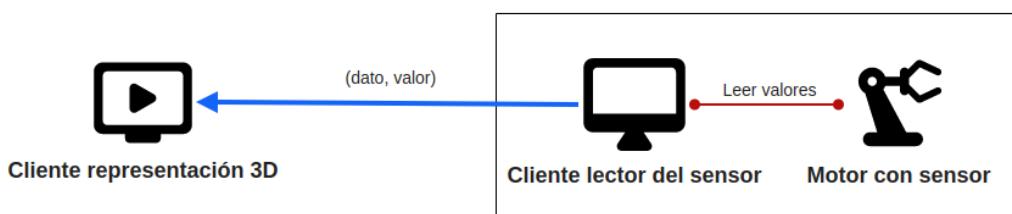


Figura 4.2: Diseño de arquitectura básica para un gemelo digital (Figura del autor)

Estos nodos se conectarán entre sí en una conexión *Point to point* (PP) utilizando paquetes que incluyen el tipo de dato enviado y su valor. El nodo que lee los valores del sensor los manda periódicamente a el nodo de representación del entorno 3D.

Esta solución incluye la característica de monitoreo remoto, pero sigue sin ser escalable y sigue siendo de alto coste en proporción con los beneficios, ya que habría que implementarlo para cada gemelo digital que se quiera utilizar.

Diseño escalable

Para lograr un diseño escalable se pueden realizar varias mejoras de muchos tipos y en diferentes partes de la arquitectura. Ahora se explicarán nuevas mejoras aplicadas sobre el diseño anterior (ver Fig.4.2) que tendrán el objetivo de mejorar la escalabilidad del sistema.

1. Un problema del diseño anterior es que no se podría crear un sistema de varios gemelos digitales, sino que tendrían que implementarse por separado. Esto se debe a que no se utiliza ninguna identificación o similar por lo que no pueden diferenciarse unos de otros.

Este problema tiene fácil solución ya que basta con incluir en los mensajes un campo que sirva como *identificador* (ver Fig.4.3).

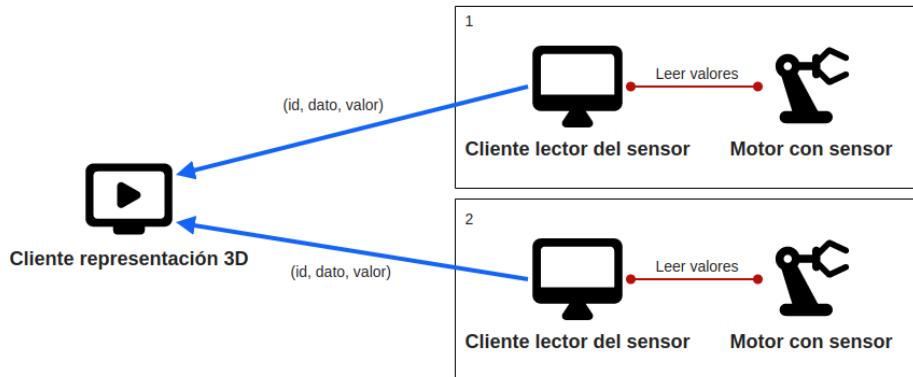


Figura 4.3: Diseño de arquitectura algo escalable (Figura del autor)

De esta forma en el entorno 3D ya se podrán visualizar varios gemelos digitales con su respectiva información.

2. Resuelto el problema de identificación siguen pudiéndose aplicar diferentes mejoras al sistema para mejorar la escalabilidad a la par que otras características.

Por ejemplo, el último diseño permite tener varios gemelos digitales, pero un único entorno 3D. Una forma de mejorar este sistema consiste en añadir un nodo que actúe como servidor. De esta forma los clientes lectores enviarán al servidor los datos junto con el identificador correspondiente y el servidor los almacenará en una base de datos. Nótese que la conexión ya no será PP sino que será *Cliente-servidor*. Por otro lado, ya se podrán utilizar varios clientes de representación 3D, los cuales pedirán al servidor los datos del gemelo que necesiten utilizando su identificador (ver Fig.4.4).

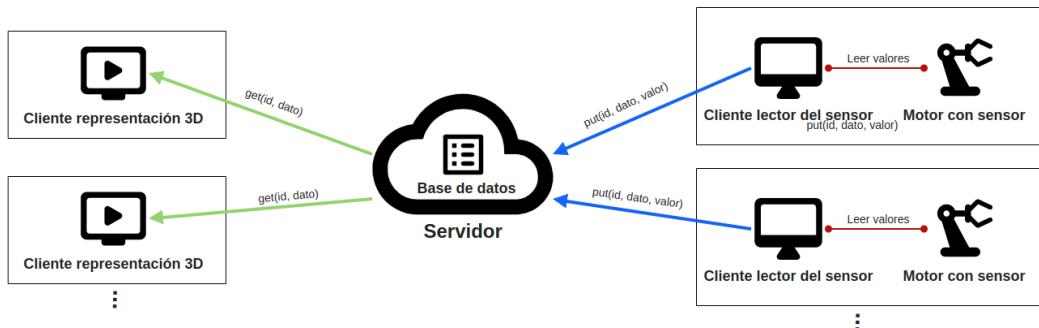


Figura 4.4: Diseño de arquitectura escalable (Figura del autor)

Con este diseño se obtienen varias ventajas:

4. RESULTADOS

- a) La primera es que se podrán tener varios entornos 3D diferentes en diferentes lugares o simplemente diferentes entornos para dividir según los gemelos digitales. Por ejemplo, en una fábrica de coches podría utilizarse un entorno 3D únicamente para las maquinas relacionadas con la fabricación y/o colocación de la carrocería, otro entorno 3D que se encargue de monitorizar las maquinas encargadas de montar los motores, etc...
- b) Otra ventaja viene dada por el uso de bases de datos. Estas permitirán tener un *historial de los estados* de cada gemelo digital además de poder acceder a grandes cantidades de datos para utilizar en cálculos o simulaciones.

Esta última arquitectura (ver Fig.4.4) ya tiene un nivel de escalabilidad adecuado y una relación coste/beneficio más positiva y será la que se utilice en el prototipo a desarrollar en el presente proyecto.

Diseño final

Si bien es cierto que ya se ha logrado obtener un diseño escalable, este todavía tiene algunos aspectos que se pueden mejorar (Ver Fig.4.5).

Un riesgo presente es que fallase el nodo servidor lo cual haría inutilizable el sistema entero. Para evitar esto pueden utilizarse varios servidores y varios brokers que funcionen como intermediarios entre los clientes y los servidores.

Al tener varios servidores sería necesario que las bases de datos estén sincronizadas, para lo que es posible añadir un componente aparte para la base de datos que se encargue de realizar esta función.

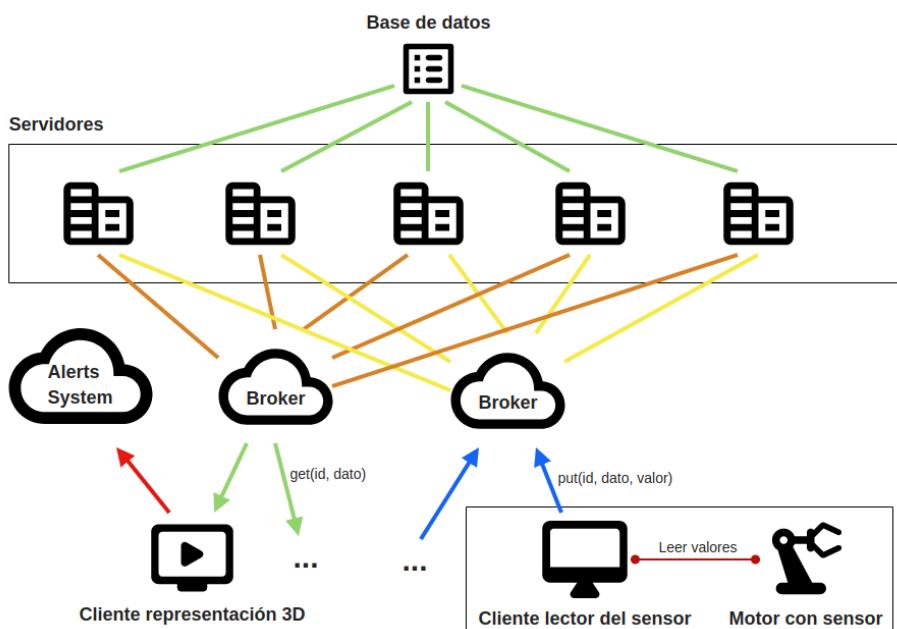


Figura 4.5: Diseño de arquitectura final (Figura del autor)

Este sistema puede hacer uso de un *sistema de alertas* existente para alertar cuando se produzca algún fallo y así evitar cuanto antes posibles daños.

Esta arquitectura aumenta la disponibilidad del sistema considerablemente, pero también aumenta el coste.

Dependiendo de las necesidades del cliente que hará uso del gemelo digital puede utilizarse esta última arquitectura, utilizar una más simple y por tanto de menos coste, o incluso si fuera necesario realizar más mejoras para aumentar la disponibilidad, seguridad, etc...

4.1.2 Diseño del modelo 3D

Para conseguir que el gemelo digital represente correctamente el estado de la parte física es necesario que el modelo 3D sea lo más fiel a la realidad posible.

Como ya se mencionó en la sección 3.4.8:Modelado 3D del capítulo 3:Metodología, se utilizará la herramienta *Blender* para construir el modelo 3D del motor eléctrico.

Existen diversas técnicas para construir un modelo 3D de un objeto real [15]:

1. *Escaneado*. Para realizar un escaneado sería necesario tener la posibilidad de hacer uso de un escaner que nos facilite esta tarea. Posteriormente en herramientas de modelado podrían corregirse imperfecciones para que el resultado sea adecuado.
2. *Fotogrametría*. Consiste tomar fotografías del objeto real desde diferentes ángulos. Posteriormente un software especializado unirá estas imágenes para crear la réplica virtual del objeto original.
3. *Rotoscopia*. La rotoscopia es una técnica que consiste en utilizar imágenes del objeto para imitar su forma en la herramienta de modelado (ver Fig.4.6). Esta técnica es útil si se dispone de los planos en 2D del objeto ya que las cámaras distorsionan el objeto dificultando la tarea.

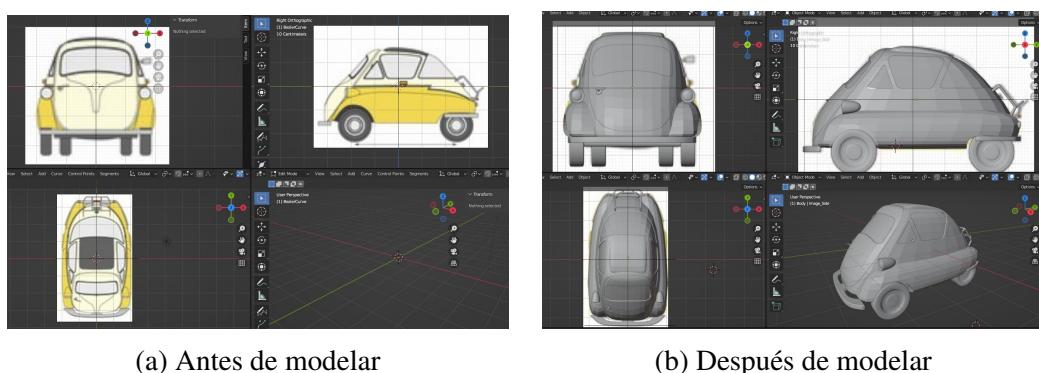


Figura 4.6: Ejemplo de rotoscopia (Fotografías tomadas por el autor)

4. *Utilizar medidas*. Una técnica sencilla es la de tomar medidas del motor para posteriormente reproducirlo en la herramienta de modelado 3D. Esta técnica es útil si el

4. RESULTADOS

Este objeto no tiene formas complejas y es fácil de medir, sin embargo, es una técnica más complicada si el objeto tiene curvas por ejemplo.

Si se poseen los planos del objeto con las medidas entonces también pueden utilizarse para modelar el objeto sin necesidad de tener que tomarlas.

Para el presente proyecto no se poseen ni un escáner que pueda ayudar en esta tarea ni los planos del objeto, por lo que se tomarán medidas del objeto manualmente para poder construir el modelo 3D en la herramienta de modelado. El resultado de las medidas se puede observar en la figura 4.7.

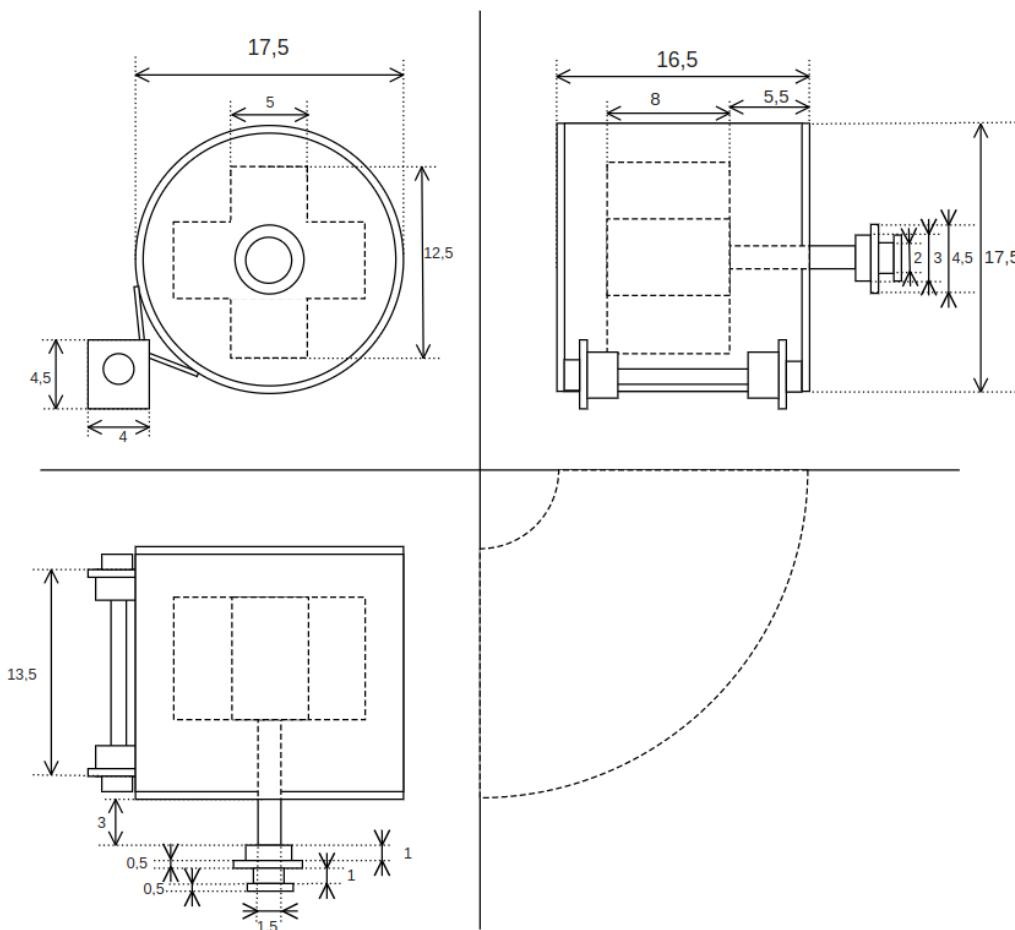


Figura 4.7: Planos del motor eléctrico (Figura del autor)

Dependiendo del nivel de detalle requerido para el gemelo digital se pueden tomar más o menos medidas, sin embargo algo que debería cumplirse siempre es que las partes móviles se modelen como piezas separadas dentro del mismo modelo. Por ejemplo, en el motor eléctrico que se utilizará en el prototipo, hay que modelar la parte rotatoria como pieza separada del resto del motor. De esta forma, en el entorno 3D se podrán representar también los movimientos de las diferentes partes del motor.

Para el prototipo del presente proyecto no es necesario un nivel de detalle elevado siempre que el motor se pueda identificar claramente a simple vista. Es por esto que no será necesario

aplicar texturas o características similares. El resultado del modelo 3D del motor eléctrico se puede observar en la figura 4.8.

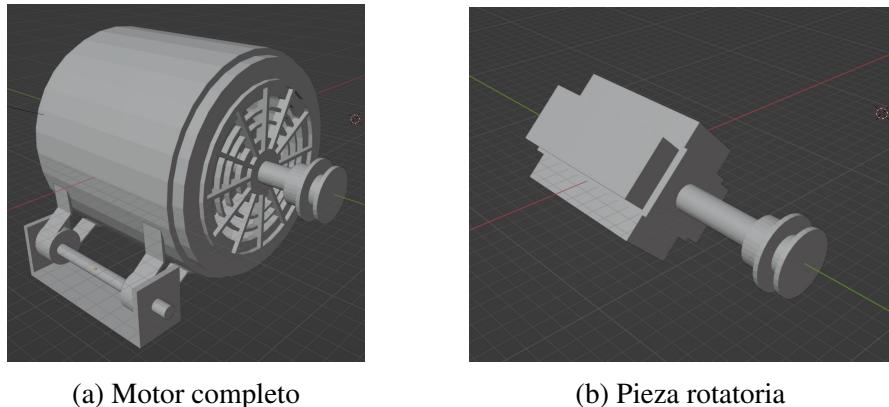


Figura 4.8: Modelo 3D del motor eléctrico (Figura del autor)

4.1.3 Diseño de la Base de datos

Como ya se ha especificado en el diseño de la arquitectura (ver sec.4.1.1) se hará uso de una base de datos que almacene los valores recogidos por los sensores además de otra información útil.

Para el diseño de la base de datos se trabajará con el objetivo de que cubra las posibles necesidades de un sistema amplio. Es decir, se diseñará teniendo en cuenta que recogerá información de diversos gemelos digitales ubicados en diferentes instalaciones ubicadas en diferentes sitios.

Se explicarán los pasos realizados desde el comienzo del diseño hasta alcanzar un diseño final adecuado.

Primer paso

Como primer paso se diseñará lo esencial para un proyecto que incluye un gemelo digital. En un proyecto de estas características lo más esencial que se almacenará serán los datos obtenidos de los sensores del motor real. Es por esto que en la base de datos será necesario tener una tabla para guardar esta información (Ver Fig.4.9).

En esta tabla será necesario tener los siguientes campos:

1. *timestamp* (k): Marca de tiempo de la lectura del valor
2. *data_type* (k): Tipo de dato que almacena (acelerómetro, temperatura, etc...)
3. *data_x*: Valor único para los tipos de datos con solo un valor (temperatura, presión, humedad y luz) y primer valor para los campos con tres valores (acelerómetro, giroscopio y magnetómetro).
4. *data_y*: Segundo valor para los tipos de datos con tres valores y nulo para el resto.

4. RESULTADOS

5. *data_z*: Tercer valor para los tipos de datos con tres valores y nulo para el resto.

Readings

timestamp	data_type	data_x	data_y	data_z
DateTime	str(20)	float	float	float

Figura 4.9: Tabla de lecturas de la base de datos (Figura del autor)

Los campos *timestamp* y *data_type* formarán la clave, lo cual no permitirá identificar una lectura, ya que no podrá existir varias lecturas en el mismo momento para el mismo tipo de datos.

Segundo paso

El diseño de la base de datos explicado hasta ahora tiene el mismo problema que tenía el diseño de la arquitectura, y es que no se pueden tener varios gemelos digitales ya que no se pueden identificar.

Ahora bien, teniendo en cuenta lo mencionado al comienzo de esta subsección, el diseño se realizará teniendo en cuenta que se tienen varias instalaciones con varios gemelos digitales cada una. Debido a esto los gemelos digitales pueden identificarse de dos maneras distintas.

La primera consiste en utilizar *identificadores globales* dentro del sistema, es decir, que cada gemelo digital utilice un único identificador que no podrá repetirse. Esta solución presenta una idea sencilla, sin embargo, no es una buena opción ya que sería muy sencillo cometer errores.

La segunda opción consiste en utilizar dos identificadores los cuales en conjunto forman un identificador global. Como se ha mencionado previamente, en el sistema puede haber varias instalaciones con varios dispositivos cada una. Teniendo en cuenta esto, cada instalación tendrá su propio identificador único que lo diferencie del resto, y cada dispositivo dentro de la instalación tendrá un identificador que lo diferencie de aquellos dentro de esa misma instalación. De esta forma utilizando estos dos identificadores de forma conjunta se referenciará a un dispositivo concreto de una instalación concreta obteniendo así una identificación global (Ver Fig.4.10).

Estos identificadores serán:

- *facility_id* : Identificador de la instalación
- *device_id* : Identificador del dispositivo dentro de la instalación

De esta forma la *clave* de la tabla de lecturas pasará a estar compuesta por los campos *facility_id*, *device_id*, *timestamp* y *data_type*.

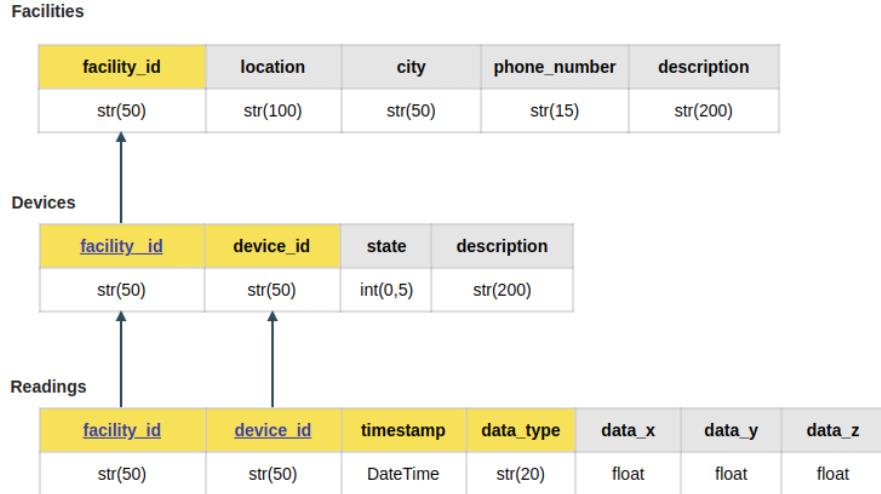


Figura 4.10: Diseño de la base de datos que permite identificación (Figura del autor)

Aprovechando estos identificadores pueden crearse tablas que almacenen información sobre las instalaciones, como por ejemplo su ubicación, ciudad, número de contacto, etc... a la par que se puede guardar información sobre los dispositivos, como por ejemplo el estado en el que se encuentran, una descripción de su funcionalidad, la ubicación dentro de las instalaciones, etc...

Tercer paso

Como se explicará más detalladamente en secciones posteriores, para cada tipo de dato de cada gemelo digital será necesario establecer unos límites o *thresholds* que nos permitan identificar cuando se están produciendo valores fuera o dentro de lo normal, permitiendo así detectar fallos lo antes posible.

Es por esto que se incluirá otra tabla que almacene esta información para cada tipo de datos de cada gemelo digital (Ver Fig.4.11).

Con este diseño escalable pueden añadirse fácilmente las tablas que sean necesarias para añadir funcionalidades según las necesidades de un proyecto sin necesidad de modificar las ya especificadas.

4.1.4 Diseño de la interfaz *ZeroC Ice*

En esta subsección se definirá la interfaz *ZeroC Ice* la cual será común para todos los nodos del sistema y por tanto se utilizará en la implementación de todos los nodos.

Para esta tarea se utilizará *slice*. *Slice* es un lenguaje que se utiliza para la definición de interfaces independientemente de su implementación, lo cual permitirá realizar invocaciones remotas¹ gracias al middleware de comunicaciones *ZeroC Ice*.

¹Invocación de una función que se ejecuta en otra máquina

4. RESULTADOS

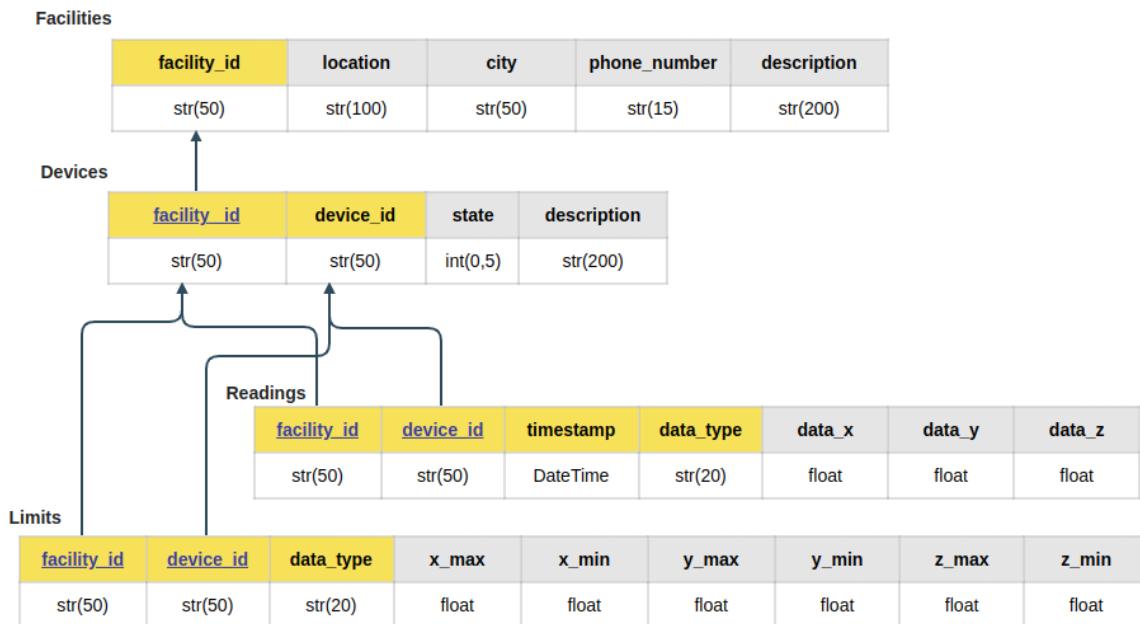


Figura 4.11: Diseño final de la base de datos incluyendo límites (Figura del autor)

Funcionalidad básica

En un primer diseño se incluirán las funciones básicas necesarias para la comunicación. Estas son las definidas en el diagrama de la arquitectura como *put* y *get*.

Los tipos de datos se han dividido en dos *categorías*: los que manejan tres valores (acelerómetro, giroscopio y magnetómetro) y los que manejan un único valor (temperatura, presión, humedad y luz) (Ver tab.4.1).

Manejan 3 valores	Manejan 1 valor
Acelerometro	Temperatura
Giroscopio	Presión
Magnetometro	Humedad
	Luz

Cuadro 4.1: Categorías de datos según los valores que manejan

Normalmente cuando se quiera acceder al valor de un dato de tres valores se querrán obtener los tres y no solo uno, por lo que para facilitar el manejo de los datos con tres valores se definirá la *estructura de datos DataSet* (Ver lst.4.1). Esta estructura estará formada por tres variables de tipo *float*.

Listado 4.1: Definición de la estructura *DataSet*

```

1 struct DataSet{
2     float x;
3     float y;
4     float z;
5 };

```

Dado que se han dividido los datos en dos categorías, es por esto que se implementarán dos funciones *put* y dos funciones *get*. De esta forma se utilizará una función *putSingleData* para trabajar con datos de valores únicos y una función *putDataSet* para trabajar con datos de tres valores. Esta misma forma de trabajo se aplicará a la función *get* quedando como resultado las funciones mostradas en el listado 4.2.

Listado 4.2: Funciones *put* y *get*

```

1 // Pull
2 DataSet getDataSetReading(string FacilityID, string DeviceID, TypeDataSet dataType) ←
   ↪ throws RegistryNotFound;
3 float getSingleDataReading(string FacilityID, string DeviceID, TypeSingleData ←
   ↪ dataType) throws RegistryNotFound;
4 // Push
5 int putDataSetReading(string FacilityID, string DeviceID, TypeDataSet dataType, ←
   ↪ DataSet data) throws InsertingError;
6 int putSingleDataReading(string FacilityID, string DeviceID, TypeSingleData ←
   ↪ dataType, float data) throws InsertingError;

```

Como parámetros se incluyen los identificadores *FacilityID* y *DeviceID* que se explicaron en la sección de diseño de la base de datos (ver sec.4.1.3) y el tipo de datos que se quiere obtener o almacenar.

Con estas funciones ya se puede implementar la *funcionalidad básica* de un gemelo digital, pero para el prototipo a desarrollar también serán necesarias funciones que nos permitan obtener y modificar los límites de los valores de cada gemelo digital.

Establecer límites

Para definir las funciones que nos permitan obtener y modificar los límites (*updateLimits* y *getLimits*) se seguirá el mismo procedimiento que el utilizado para las funciones *put* y *get*.

En primer lugar, dado que por cada valor es necesario establecer límites inferiores y superiores, se definirán estructuras de datos que faciliten el manejo de la información.

Para los datos de un único valor se definirá la estructura *SingleDataLimits* la cual estará formada por dos variables de tipo *float* (Ver lst.4.3). De esta manera almacena tanto el límite superior como el inferior.

Del mismo modo, para los datos que manejan tres valores, se define la estructura *DataSetLimits* que estará formada por seis variables de tipo *float* (Ver lst.4.3).

Listado 4.3: Definición de la estructuras *SingleDataLimits* y *DataSetLimits*

```

1 struct SingleDataLimits{
2     float maxLimit;
3     float minLimit;

```

4. RESULTADOS

```
4 };
5
6 struct DataSetLimits{
7     float xMaxLimit;
8     float xMinLimit;
9     float yMaxLimit;
10    float yMinLimit;
11    float zMaxLimit;
12    float zMinLimit;
13 }
```

Una vez definidas las estructura que facilitarán el manejo de los datos se definirán las funciones que nos permitirán consultar y modificar los valores de los límites. De nuevo se definirán funciones por separado para los datos de valor único y los datos de tres valores. El resultado de estas funciones se muestra en el listado 4.4.

Listado 4.4: Funciones *updateLimits* y *getLimits*

```
1 // Get
2 DataSetLimits getDataSetLimits(string FacilityID, string DeviceID, TypeDataSet ←
3     ↪ dataType) throws RegistryNotFound;
4 SingleDataLimits getSingleDataLimits(string FacilityID, string DeviceID, ←
5     ↪ TypeSingleData dataType) throws RegistryNotFound;
6 // Update
7 int updateDataSetLimits(string FacilityID, string DeviceID, TypeDataSet dataType, ←
8     ↪ DataSetLimits limits) throws InsertingError;
9 int updateSingleDataLimits(string FacilityID, string DeviceID, TypeSingleData ←
10    ↪ dataType, SingleDataLimits limits) throws InsertingError;
```

En este caso los parámetros cumplen la misma función que en las funciones definidas anteriormente.

Funcionalidades extra

En caso de que se quieran definir otras funcionalidades simplemente habría que seguir el mismo proceso que en los dos pasos anteriores.

Primero habría que definir estructuras en caso de que fuese necesario, de tal manera que nos faciliten el manejo de datos. Tras esto simplemente se definen las funciones con los parámetros que sean necesarios.

Un ejemplo simple sería definir funciones que permitan obtener la media y la desviación estándar de los valores en los últimos segundos. Para este ejemplo no sería necesario definir estructuras nuevas ya que basta con utilizar la estructura *DataSet* (4.1) definida previamente.

4.2 Implementación

En esta sección se describirá detalladamente el proceso de implementación de los diferentes sub-objetivos del presente proyecto. El orden de implementación de los sub-objetivos no es necesariamente el orden en el que se explicarán en el presente documento. De hecho los diferentes sub-objetivos se han ido implementando de manera paralela aplicando el modelo en espiral mencionado en el capítulo 3:Metodología de tal forma que permitiera realizar pruebas de integración del sistema del prototipo a medida que se implementaban nuevas funciones.

4.2.1 Implementación del sub-objetivo 1

El primer sub-objetivo consiste en diseñar y desarrollar un software que recopile y almacene los datos y el estado de determinados recursos físicos (ej. motor eléctrico) además de proveer esta información cuando se solicite.

Para cumplir este objetivo se ha decidido que se implementará un software que actúe como servidor que será integrado en la nube.

Para desarrollar este servidor se utilizará:

- El lenguaje de programación *Python* (versión *Python 3*).
- El middleware *ZeroZ Ice*.
- El administrador de bases de datos *Postgresql*.

En las siguientes secciones se explicarán las diferentes partes relevantes en el desarrollo. De esta forma siguiendo el proceso se podrá conseguir un gemelo digital cuyas funcionalidades podrán ser extendidas en caso de ser necesario.

Archivo de configuración

En este caso se utilizará un archivo de configuración para poder modificar fácilmente algunos campos sin tener que modificar el código.

En este caso se ha decidido usar el paquete *configparser* [19] ya que facilita en gran medida esta tarea. Es recomendable que dicho archivo de configuración tenga al menos los campos especificados en el listado 4.5, siendo posible añadir tantos como se quiera.

Listado 4.5: Archivo de configuración del servidor

```

1 [database]
2 address = postgresql://tfg_history
3
4 [zeroc-ice]
5 adapter.name = DigitalTwinAdapter
6 adapter.endpoint = default -p 10000
7 identity = DigitalTwinServer

```

4. RESULTADOS

Uso de la base de datos

Para hacer uso de la base de datos se utilizará el paquete *sqlalchemy* [23] ya que facilita mucho esta tarea.

En primer lugar habrá que conectarse a la base de datos a través de la dirección establecida (línea 1) en el archivo de configuración, entonces se podrá crear una sesión (líneas 2 y 3) para empezar a trabajar con las tablas y sus registros (ver lst.4.6).

Listado 4.6: Inicio de sesión en la base de datos

```
1 engine = create_engine(DB_ADDR)
2 Session = sessionmaker(engine)
3 session = Session()
```

El paquete *sqlalchemy* permite trabajar con objetos, por lo que para poder trabajar con las tablas es necesario definirlas primero.

Para definir una tabla se crea una clase que herede de la clase *Base* y se especifica el nombre que se le quiere dar a la tabla en el atributo denominado *__tablename__*. Acto seguido se definen los campos que tendrán dichas tablas indicando características como el tipo de dato, la clave primaria, valores por defecto, etc... (ver lst.4.7).

Listado 4.7: Definición de las tablas con *sqlalchemy*

```
1 Base = declarative_base()
2
3 class DataReading(Base):
4     # Table name
5     __tablename__ = 'Readings'
6
7     # Attributes
8     facility_id = Column(String(50), primary_key=True)
9     device_id = Column(String(50), primary_key=True)
10    timestamp = Column(DateTime(), primary_key=True, default=datetime.now())
11    data_type = Column(String(20), primary_key=True)
12    data_x = Column(Float(), nullable=False)
13    data_y = Column(Float(), nullable=True, default=None)
14    data_z = Column(Float(), nullable=True, default=None)
15
16    def __str__(self):
17        return self.facility_id + "," + self.device_id + "," + ↵
18            str(self.timestamp) + "," + self.data_type + "," + ↵
19            str(self.data_x) + "," + str(self.data_y) + "," + ↵
20            str(self.data_z) + "]"
```

Implementar funciones

El siguiente paso continuar con la implementación consiste implementar las funciones definidas en la interfaz *ZeroC Ice* en generar los stubs en lenguaje *Python* a partir del archivo de la interfaz *ZeroC Ice* por lo que hay que ejecutar el siguiente comando:

```
$ slice2py <directorio>/dt.ice
GNU/Linux
```

Esto generará automáticamente el código *python* necesario. Tras esto ya se puede importar el módulo definido en la interfaz e implementar las funciones definidas para posteriormente poder invocarlas.

Para definir las funciones es necesario definir la clase que implementa la interfaz. A esta clase en este proyecto se le llamará *dataSinkI*. Tras esto, las funciones de la interfaz podrán definirse como funciones de esta clase.

1. Funciones *get()*. La idea detrás de la implementación es sencilla (ver lst.4.8).

En primer lugar, habrá que realizar una petición de lectura a la base de datos (línea 8) aplicando varios filtros. Se filtrará por *FacilityID* y *DeviceID* para identificar el motor y se filtrará por *data_type* para seleccionar el tipo de datos requerido (líneas 9, 10 y 11 respectivamente). Tras esto habrá que ordenar las entradas según el *timestamp* (línea 12) para obtener la más reciente.

Si se ha encontrado una entrada se devolverá el valor o conjunto de valores de esta entrada, y si no, se lanzará una excepción.

Listado 4.8: Implementación de las funciones *get*

```

1 # ZeroC-Ice server
2 class dataSinkI(digitaltwin.dataSink):
3
4     # Pull
5
6     def getDataSetReading(self, FacilityID, DeviceID, dataType, current=None):
7         logging.info("getDataSetReading_[ "+FacilityID+", "+DeviceID+", "
8                     " "+dataType.name+" ]")
9
10    reading = session.query(models.DataReading
11                            .filter(models.DataReading.facility_id==FacilityID)
12                            .filter(models.DataReading.device_id==DeviceID)
13                            .filter(models.DataReading.data_type==dataType.name)
14                            .order_by(desc(models.DataReading.timestamp)).first()
15
16    if reading:
17        return digitaltwin.DataSet(reading.data_x, reading.data_y,
18                                    reading.data_z)
19    else: # No reading found
20        logging.warning("No_registry_found")
```

4. RESULTADOS

```
18     raise digitaltwin.RegistryNotFound
19
20     def getSingleDataReading(self, FacilityID, DeviceID, dataType, ↵
21         ↵ current=None):
22         logging.info("getSingleDataReading["++FacilityID+, "+DeviceID+, ↵
23             ↵ "+dataType.name+"]")
24
25         reading = session.query(models.DataReading
26             .filter(models.DataReading.facility_id==FacilityID
27             .filter(models.DataReading.device_id==DeviceID
28             .filter(models.DataReading.data_type==dataType.name
29             .order_by(desc(models.DataReading.timestamp)).first()
30
31         if reading:
32             return reading.data_x
33         else: # No reading found
34             logging.warning("No_registry_found")
35             raise digitaltwin.RegistryNotFound
```

2. Funciones *put()*. En el caso de las funciones *put* la implementación es incluso más sencilla (ver lst.4.9).

Como se ha mencionado previamente, el paquete *sqlalchemy* permite trabajar con objetos, por lo que primero habrá que crear un objeto de la entrada que se pretende escribir con los argumentos introducidos (línea 5).

Una vez creado habrá que añadir la entrada al *stack* de la sesión (línea 7) y guardar los cambios en la base de datos (línea 9). En caso de que ocurriese un error se lanzará una excepción.

Listado 4.9: Implementación de las funciones *put*

```
1 def putDataSetReading(self, FacilityID, DeviceID, dataType, data, current=None):
2     logging.info("putDataSetReading["++FacilityID+, "+DeviceID+, ↵
3         ↵ "+dataType.name+", "+ str(data) + "]")
4
5     moment = datetime.datetime.now()
6     reading = models.DataReading(facility_id=FacilityID, device_id=DeviceID, ↵
7         ↵ timestamp=moment, data_type=dataType.name, data_x=data.x, ↵
8         ↵ data_y=data.y, data_z=data.z)
9
10    session.add(reading)
11    try:
12        session.commit()
13    except:
14        raise digitaltwin.InsertingError
15
16    return 1
```

```

14
15 def putSingleDataReading(self, FacilityID, DeviceID, dataType, data, ↵
16     ↵ current=None):
17     logging.info("putSingleDataReading_[ "+FacilityID+" , "+DeviceID+", ↵
18         ↵ "+dataType.name+", "+ str(data) + "]")
19
20     moment = datetime.datetime.now()
21     reading = models.DataReading(facility_id=FacilityID, device_id=DeviceID, ↵
22         ↵ timestamp=moment, data_type=dataType.name, data_x=data)
23
24     session.add(reading)
25     try:
26         session.commit()
27     except:
28         raise digitaltwin.InsertingError
29
30
31     return 1

```

3. Funciones *getLimits()*. La implementación de las funciones que permiten obtener los límites seguirá el mismo proceso que las funciones *get* con la única diferencia de que trabajará con la tabla *Limits*.
4. Funciones *updateLimits()*. Esta implementación es similar a la de las funciones *put*, sin embargo, varía un poco (ver lst.4.10).

En lugar de tratar de escribir una nueva entrada, se actualizará una existente, de modo que siempre habrá como máximo una entrada para cada tipo de datos de cada gemelo digital.

Es por esto que primero se leerá la entrada de límites para el gemelo digital y tipo de datos especificado en los argumentos. Una vez obtenida, se modificarán los límites según los valores introducidos en los argumentos y se guardarán los cambios.

En caso de que no exista, se creará un objeto nuevo y se escribirá una entrada nueva.

Listado 4.10: Implementación de las funciones *updateLimits*

```

1 def updateDataSetLimits(self, FacilityID, DeviceID, dataType, limits, ↵
2     ↵ current=None):...
3
4 def updateSingleDataLimits(self, FacilityID, DeviceID, dataType, limits, ↵
5     ↵ current=None):
6     logging.info("updateSingleDataLimits_[ "+FacilityID+" , "+DeviceID+", ↵
7         ↵ "+dataType.name+"]")
8
9     limit = session.query(models.Limits
10         .filter(models.Limits.facility_id==FacilityID
11             .filter(models.Limits.device_id==DeviceID

```

4. RESULTADOS

```
9         ).filter(models.Limits.data_type==dataType.name
10             ).first()
11
12     if limit:  # Limit found
13         limit.x_max = limits.maxLimit
14         limit.x_min = limits.minLimit
15         session.add(limit)
16         try:
17             session.commit()
18         except:
19             raise digitaltwin.InsertingError
20         return 1
21
22     else:    # No limit found
23         limit = models.Limits(facility_id=FacilityID, device_id=DeviceID, ↵
24             ↵ data_type=dataType.name, x_max=limits.maxLimit, x_min = ↵
25             ↵ limits.minLimit)
26
27         session.add(limit)
28         try:
29             session.commit()
30         except:
31             raise digitaltwin.InsertingError
32         return 1
```

Inicializar entorno Ice

Por último, para que otros nodos puedan realizar invocaciones remotas de las funciones implementadas es necesario inicializar el entorno *ZeroC Ice* (ver lst.4.11).

Para esta tarea primero será necesario crear una instancia de la clase *dataSinkI* definida anteriormente (línea 2). Tras esto se creará un adaptador con el nombre y el *endpoint* definidos en el archivo de configuración (línea 4). Una vez creado el adaptador se añade el objeto creado previamente con la identidad especificada en el archivo de configuración (línea 5).

Una vez añadido solo queda activar el adaptador (línea 6) y activar el bucle de eventos hasta que se finalice el proceso (línea 7).

Listado 4.11: Inicialización del entorno *ZeroC Ice* del servidor

```
1 with Ice.initialize(sys.argv) as communicator:
2     object = dataSinkI()
3
4     adapter = communicator.createObjectAdapterWithEndpoints(ZICE_ADAP_NAME, ↵
5         ↵ ZICE_APAD_ENDPOINT)
6     adapter.add(object, communicator.stringToIdentity(ZICE_IDENTITY))
7     adapter.activate()
8     communicator.waitForShutdown()
```

Integración en la nube

El último paso para concluir el primer sub-objetivo consiste en implementar el servidor en la nube.

Para esta tarea se utilizará la herramienta *Docker* [7]. Esta herramienta permite crear *contenedores*² que pueden ser ejecutados simulando una máquina real de forma similar a la que lo hace una máquina virtual.

Para esta tarea simplemente habrá que crear un archivo *Dockerfile* que será el encargado de preparar y construir el contenedor para su ejecución (ver lst.4.12). Se pueden crear varios archivos *Dockerfile* para diferentes nodos, es decir, si se quiere tener un nodo independiente para la base de datos se tendrá que utilizar otro archivo *Dockerfile* distinto.

Listado 4.12: Archivo Dockerfile del servidor

```

1 FROM python:3
2 ADD *.py /
3 ADD digitaltwin/*.py /digitaltwin/
4 ADD dt.config /
5 ADD requirements.txt /
6 RUN pip install -r requirements.txt
7 CMD [ "python3", "./dt.py" ]

```

Una vez creados los archivos *Dockerfile* habrá que ejecutar los siguientes comandos con las opciones convenientes para construir los contenedores:

```

$ docker build --rm -f dockerfiles/Dockerfile.dt --tag dt-server .
$ docker build --rm -f dockerfiles/Dockerfile.db --tag dt-database .
GNU/Linux

```

Tras construir los contenedores estos podrán ejecutarse utilizando los siguientes comandos con las opciones convenientes (nótese que el contenedor que alojará la base de datos se lanzará el primero):

```

$ docker run --rm -ti --name db --hostname db -p 5432:5432 dt-database
$ docker run --rm -ti --name server --hostname server -p 10000:10000 dt-server
GNU/Linux

```

Una vez construidos los contenedores estos podrán ejecutarse donde se haya considerado según el proyecto, ya sea en un servidor propio, haciendo uso de un servicio de ejecución en la nube, etcétera...

Para el presente proyecto el docker se lanzará en el mismo portátil en el que se lanzará el cliente de representación 3D. Dado que el funcionamiento de un contenedor *docker* es similar al de una máquina virtual, servirá de la misma forma que si se estuviera ejecutando en una máquina externa.

²Unidades estandarizadas que incluyen todo lo necesario para ejecutarse

4. RESULTADOS

4.2.2 Implementación del sub-objetivo 2

El segundo sub-objetivo consiste en diseñar y desarrollar un software que muestre una interfaz de visualización 3D con capacidad de predicción/simulación y representación de la información asociada a un recurso físico.

Para cumplir esta función se desarrollará un software cliente que realice la representación 3D del motor eléctrico y muestre los datos almacenados y/o simulados por el software servidor anterior.

Las herramientas que se utilizarán para implementar este sub-objetivo son:

1. El software *Unity* para crear el entorno 3D.
2. El lenguaje de programación *C#* ya que es el que usa *Unity*.
3. El middleware *ZeroC Ice*.

La herramienta Unity sigue el modelo *Object-Oriented Programming* (OOP) cuya unidad más básica son los denominados *Game Object*. Cada *game object* posee unas propiedades que definen su aspecto y su comportamiento en el entorno 3D. Ahora bien, dentro de estas propiedades o componentes existen varios tipos.

Para el presente proyecto resulta de interés los componentes *Script* los cuales son archivos de código que el objeto ejecutará, y por tanto, puede definir alguna funcionalidad de dicho objeto. Un ejemplo claro dentro del presente proyecto sería la utilización de un *script* que se encargue de conectarse al servidor que se especifique.

Dado que se utilizan una gran cantidad de componentes, en el presente documento se explicarán aquellos componentes más relevantes para este proyecto.

Entorno *ZeroC Ice*

La primera tarea de este sub-objetivo consistirá en preparar el entorno *ZeroC Ice* ya que no es tan sencillo como con otras herramientas. Esto se debe a que *ZeroC Ice* no es un paquete que se pueda instalar desde el instalador de paquetes incluido en la herramienta *Unity*. Por tanto para poder instalar el paquete primero será necesario instalar otro manejador de paquetes llamado *NuGet* (se recomienda seguir las instrucciones de instalación que aparecen en el repositorio [16]).

Tras haber instalado el paquete *Nuget* este se utilizará para instalar el paquete *ZeroC Ice* (buscar como *zeroc.ice.net*).

Una vez instalado el paquete se puede proceder a inicializar el entorno 3D. Primero habrá que generar los *stubs* a partir de la interfaz *ZeroC Ice* para lo que habrá que ejecutar el siguiente comando, el cual generará automáticamente el código python necesario:

```
$ slice2cs <siretorio>/dt.ice --output-dir <directorio>/DigitalTwinMonitor/Assets/Scripts  
GNU/Linux
```

Una vez generado el código ya se puede importar el módulo correspondiente al gemelo digital e inicializar el entorno (ver lst.4.13). Primero se obtiene el objeto *communicator* el cual permite usar la funcionalidad del *broker* (línea 5). Tras esto habrá que obtener el proxy a partir de la cadena introducida (línea 10) y a partir de este proxy obtener el objeto que hace referencia al servidor (línea 12). Si no se produce ningún error ya se podrán realizar las invocaciones remotas de las funciones del servidor para así obtener los valores necesarios.

Listado 4.13: Inicialización del entorno *ZeroC Ice* del cliente de representación 3D

```

1  using digitaltwin;
2
3  public void Connect(){
4
5      dt_client.communicator = Ice.Util.initialize();
6
7      if (dt_client.communicator == null){
8          Debug.LogError("Fail!, getting communicator!");
9      }else{
10         this.obj = communicator.stringToProxy(proxyString);
11         try{
12             this.digitalTwin = dataSinkPrxHelper.checkedCast(this.obj);
13
14             obtenerValores();
15
16         }catch{
17             Debug.LogError("Error while getting washmachine Prx\n");
18         }
19     }
20 }
```

En este script pueden utilizarse variables para permitirnos controlar diferentes factores, como por ejemplo, la cadena del proxy al que se conectará, los identificadores de la instalación y el dispositivo del gemelo digital que representa o que tipos de datos se sincronizarán y cada cuando tiempo... (ver Fig.4.12).

Gráficas de valores

Para el presente proyecto será necesario representar gráficamente los valores de los sensores. La herramienta *Unity* no dispone de un componente nativo que permita realizar esta tarea por lo que se tienen dos opciones: la primera opción consistiría en buscar un paquete que nos ayude con esta función en la *Asset Store* [26], y la segunda opción consiste en implementar este componente desde cero.

En la *Asset Store* existen una gran variedad de componentes de todo tipo que se pueden comprar para usar en tus proyectos. También existen otros componentes gratis que se pueden descargar, pero en caso de usar alguno habrá que asegurarse de conocer y leer la normativa

4. RESULTADOS

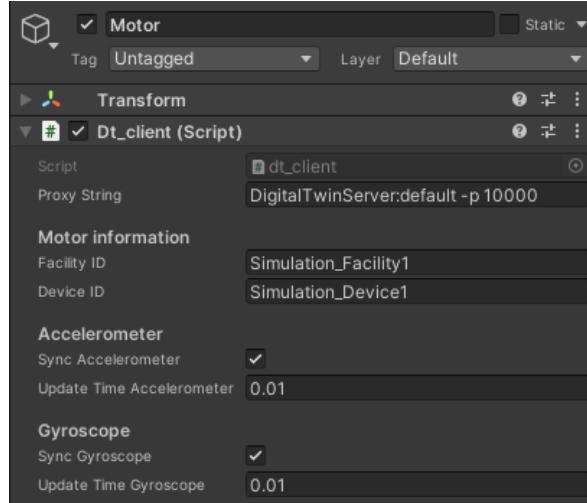


Figura 4.12: Componente *script* del motor que se conecta a un servidor (Figura del autor)

de uso de estos componentes. Dado que no se ha encontrado un componente que se acomode del todo a las necesidades de este proyecto se implementará uno desde cero inspirado en un componente explicado en el canal de *Youtube Code Monkey* [5].

El componente estará formado por dos *scripts* que se encargarán de diferentes tareas además de permitir configurar diferentes características sin necesidad de cambiar el código (ver fig.4.13).

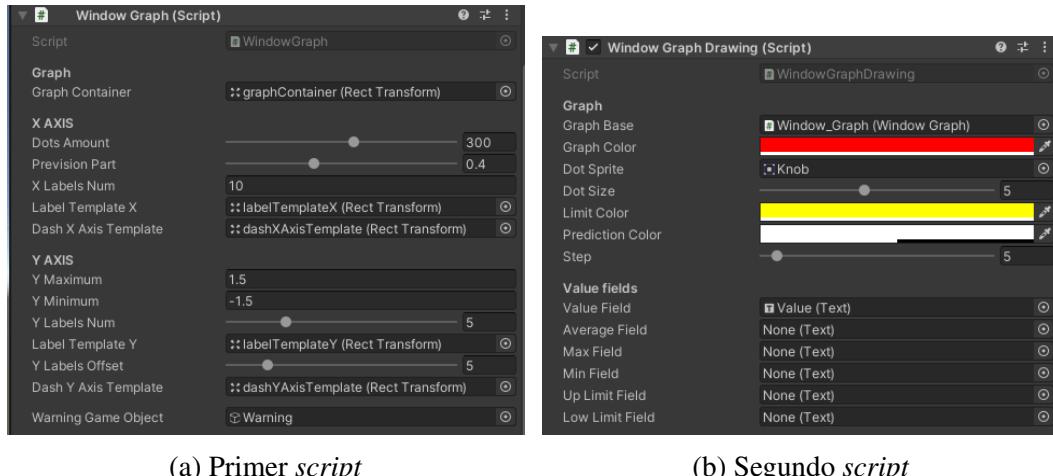


Figura 4.13: *Scripts* del componente que muestra una gráfica de valores (Figura del autor)

El primero de estos *scripts* se encargará de dibujar la estructura de la gráfica, es decir, las líneas separadoras y las etiquetas o *labels* que indican los valores que representa la gráfica.

El segundo *script* se encargará de dibujar en la gráfica los valores recibidos y los límites entre otras funciones.

El resultado se muestra en la figura 4.14.

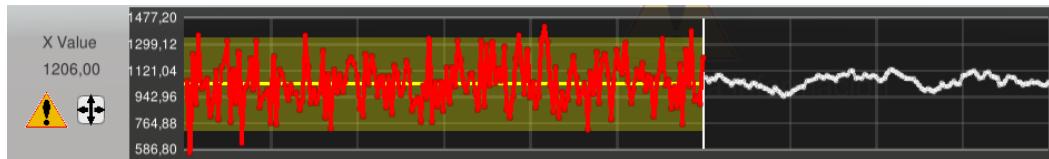


Figura 4.14: Componente gráfica de valores (Figura del autor)

Algunas funciones añadidas a la representación de valores podrían ser la representación de valores simulados (valores de color blanco de la parte derecha), la alerta al usuario en caso de que se sobrepase algún límite establecido, ajustar la gráfica, etc...

Pestañas de datos

Una vez definido el componente este se puede usar tantas veces como sea necesario e incluso utilizarlo dentro de otros componentes. Dado que hay dos categorías de datos según el número valores, como se definió en el capítulo 3:Metodología, se definirán dos nuevos componentes adecuados a cada tipo de datos.

Para datos con tres valores como puede ser el acelerómetro basta con construir un nuevo objeto con tres componentes gráficos como el creado recientemente (ver fig.4.15).

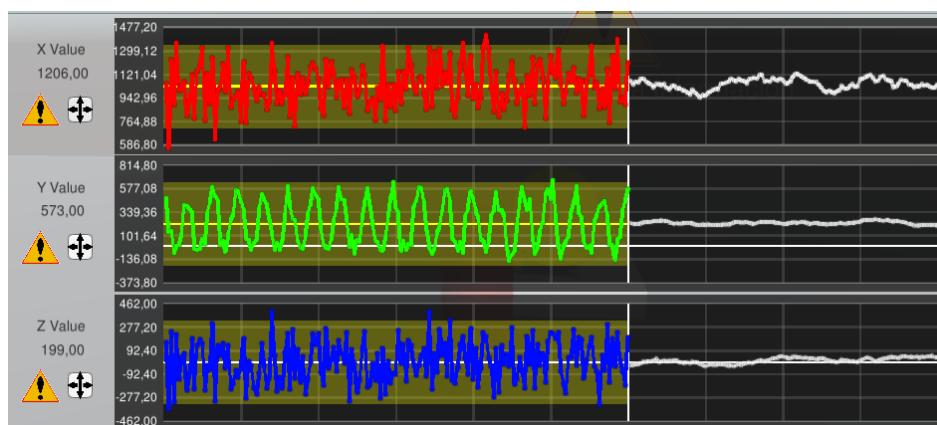


Figura 4.15: Objeto de representación de datos de tres valores (Figura del autor)

Por otro lado para los datos de un único valor basta con incluir el componente una única vez (ver fig.4.16). Al hueco restante puede utilizarse para mostrar información extra como por ejemplo el valor máximo, el valor mínimo, los valores exactos de los límites, etc...

Ventana de información

Una vez creados algunos objetos auxiliares para representar la información se puede construir la ventana de información del gemelo digital.

Se implementará una ventana que muestre toda la información del gemelo digital y permita interactuar con este de diferentes maneras. En el presente proyecto se implementará para que muestre los identificadores de la instalación y el dispositivo que representa, el proxy al que se

4. RESULTADOS

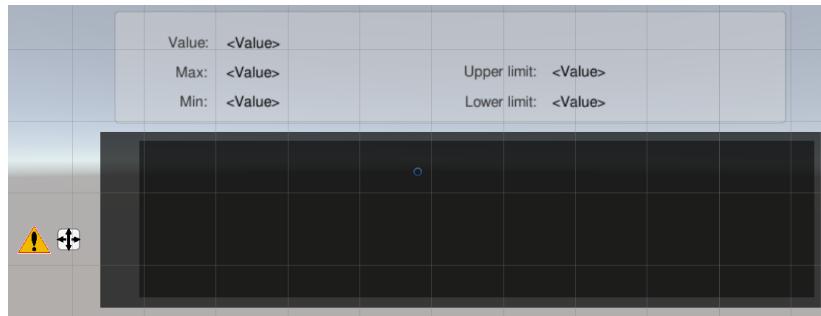


Figura 4.16: Objeto de representación de datos de un valor (Figura del autor)

conecta para obtener los datos, el estado de la conexión y los valores recibidos (ver fig.4.17). Cabe mencionar que esto siempre puede adaptarse según las necesidades del proyecto.



Figura 4.17: Ventana de información del gemelo digital (Figura del autor)

Se permite modificar tanto los identificadores como el proxy de tal manera que se puede cambiar en cualquier momento el servidor al que se conecta y el gemelo digital que representa.

Interfaz del usuario

El control del entorno 3D puede realizarse de diferentes maneras según las especificaciones de cada proyecto. Por ejemplo, una posible implementación sería controlar una cámara flotante y seleccionar el gemelo digital con el ratón, otra posible opción consistiría en tener una lista de los gemelos y al seleccionar uno hacer que este se muestre en 3D a la par que los datos en las gráficas.

En este proyecto se ha optado por una implementación que se aproxime más a lo que sería un entorno real. Esto quiere decir que se controlará un personaje a través de una instalación

recreada en el entorno 3D. Con esta implementación pueden identificarse de mejor manera las localizaciones de los diferentes dispositivos en el entorno real (ver fig.4.18).

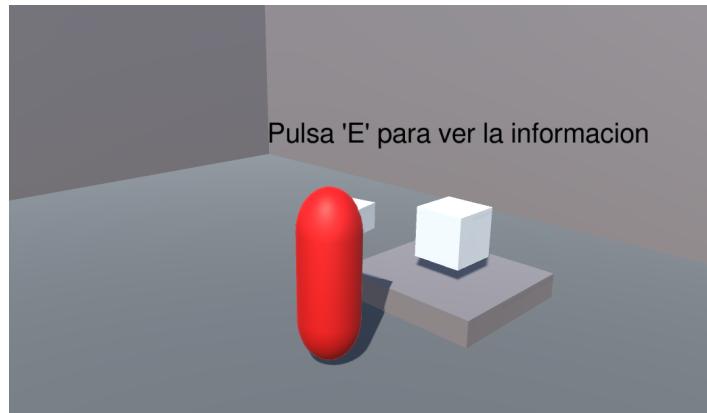


Figura 4.18: Interfaz del usuario en el entorno 3D (Figura del autor)

La mejor implementación sería aquella que mejor se adapte a las necesidades concretas de cada proyecto, adaptando el nivel de detalle de igual manera.

Dado que existe un entorno 3D, la ventana diseñada anteriormente puede mostrarse en 3D próxima al correspondiente gemelo digital de tal forma que se pueda ver la información de varios gemelos digitales cercanos a la vez (ver fig.4.19). De esta forma se aprovechará mejor el entorno 3D.

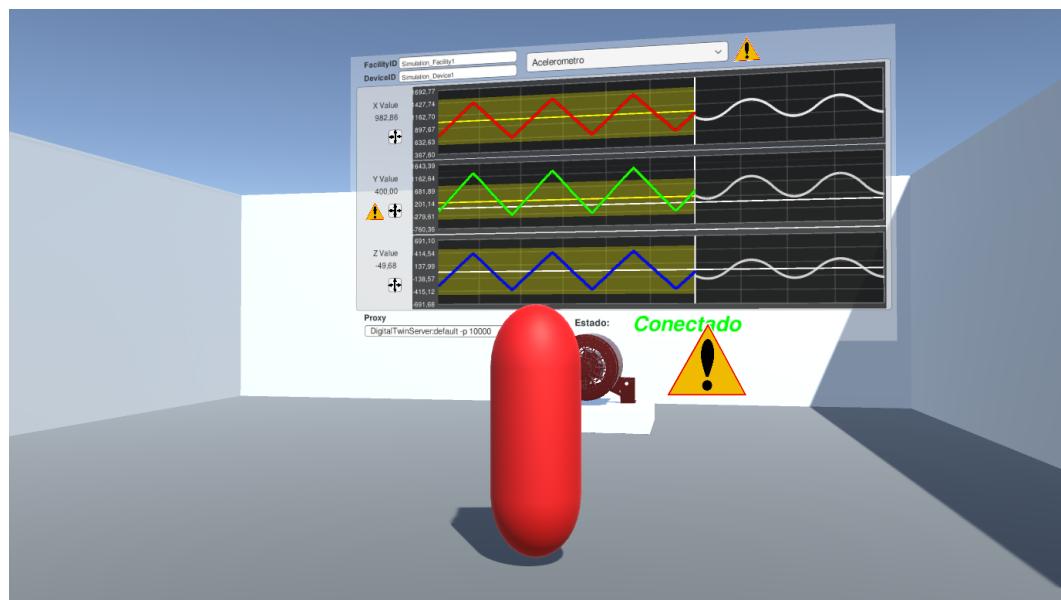


Figura 4.19: Ventana de información en 3D (Figura del autor)

Simulación

Como ya se ha explicado, una de las ventajas de los gemelos digitales es la capacidad de simulación con diferentes objetivos, ya sea para probar un diseño antes de implementarlo,

4. RESULTADOS

comprobar la reacción del sistema en diferentes circunstancias, para realizar un mantenimiento predictivo, etc...

La simulación es un trabajo costoso que dependerá de las necesidades y los objetivos de cada proyecto, así como de los recursos. Para el presente proyecto se implementará una solución sencilla orientada al mantenimiento predictivo.

La simulación consistirá en que la gráfica muestre una serie de valores esperados en función de los valores recibidos (ver fig.4.20).

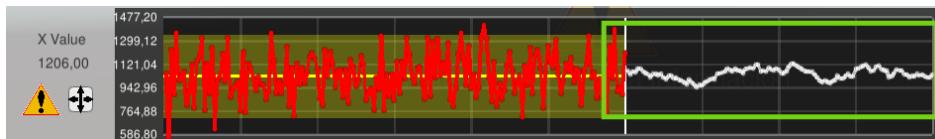


Figura 4.20: Valores simulados en las gráficas del entorno 3D (Figura del autor)

Existen diversas técnicas de simulación con distintos grados de dificultad y/o coste por lo que habría que analizar cual se adecúa mejor al proyecto a desarrollar, ya que la simulación, sobre todo en tiempo real, requiere de cierta capacidad computacional.

Para este proyecto se utilizará una técnica sencilla debido a los limitados recursos de tiempo y de procesamiento. La técnica a utilizar consiste en lo siguiente, se obtendrá al *media móvil*³ de los últimos segundos y estos serán los datos que se esperan por lo que se mostrarán a la derecha.

El resultado de los valores simulados (color blanco en la figura 4.20) no mostrarán los valores esperados como tal, sino el dibujo o trazado que se espera que muestren los valores futuros, es decir, si la parte simulada muestra un aumento constante no significa que se esperan esos valores concretos, sino que se espera que los valores futuros tengan un aumento constante.

4.2.3 Implementación del sub-objetivo 3

El tercer sub-objetivo consiste en el diseño y desarrollo de un software que obtenga datos de los recursos físicos (ej. motor eléctrico).

Para cumplir esta función se desarrollará un cliente que recopila la información de los sensores reales conectados a un motor eléctrico real y los envía al software servidor mencionado anteriormente para que los almacene.

Para el desarrollo de este sub-objetivo se utilizará:

- El lenguaje de programación *Python* (versión *Python 3*).
- El middleware *ZeroZ Ice*.

³Cálculo utilizado para analizar un conjunto de datos y obtener un conjunto de promedios

Archivo de configuración

En este nodo también se hará uso de archivos de configuración, en concreto uno con el formato de los archivos de configuración de *Ice* (ver lst.4.14). Algunos atributos que guardará son:

1. *dt.Server* → Dirección del servidor al que se conectará.
2. *FacilityID* → Identificador de la instalación en la que se encuentra.
3. *DeviceID* → Identificador del dispositivo dentro de la instalación (debe ser único entre los dispositivos de la misma instalación, pero puede repetirse con otro dispositivo de otra instalación).

Haciendo uso de los dos identificadores se tiene un identificador global único para el dispositivo que se utilice.

Listado 4.14: Archivo de configuración del cliente que lee los sensores

```

1 dt.Server=DigitalTwinServer:default -p 10000
2
3 FacilityID=Test Facility 1
4 DeviceID=Test Device 1

```

Inicializar entorno *ZeroC Ice*

Para poder iniciar el entorno *ZeroC Ice* correctamente habrá crear los stubs correspondientes a la interfaz de gemelo digital. Dado que el lenguaje de programación utilizado en este nodo es *python*, se utilizará el siguiente comando, el cual generará automáticamente el código *python* necesario:

```
$ slice2py <directorio>/dt.ice
GNU/Linux
```

Para poder conectarse al servidor e invocar remotamente sus funciones es necesario inicializar el entorno (ver lst.4.15).

Para esto será necesario crear una clase *hija* de la clase *Ice.Application* e implementar el método *run*. En este método se obtendrán los atributos del archivo de configuración explicado previamente (líneas 7, 8 y 9) obteniendo un proxy a partir de la dirección especificada. Tras esto se obtiene una referencia al objeto remoto que representa al servidor (línea 11). Una vez hecho esto ya se podrán realizar invocaciones remotas al servidor.

Listado 4.15: Inicialización del entorno *ZeroC Ice* del cliente lector de sensores

```

1 class gw(Ice.Application):
2     FacilityID = None
3     DeviceID = None
4

```

4. RESULTADOS

```
5     def run(self, argv):
6         properties = self.communicator().getProperties()
7         proxy = self.communicator().stringToProxy(properties.getProperty("dt.Server"))
8         self.FacilityID = properties.getProperty("FacilityID")
9         self.DeviceID = properties.getProperty("DeviceID")
10
11        dt = digitaltwin.dataSinkPrx.checkedCast(proxy)
12        if not dt:
13            raise RuntimeError("Invalid_proxy")
14
15        # Leer y enviar los datos al servidor
16        while 1:
17            leerSensores()
18            enviarDatos()
```

Envío de datos

Una vez obtenida la referencia al servidor solamente queda leer los datos de los sensores usados y mandarlos.

La forma en que se leen los sensores dependerá del sensor que se utiliza y la implementación que se realice en cada proyecto ya que existen varias maneras diferentes de llevar a cabo esta tarea.

Para el envío de datos sería altamente recomendable utilizar diferentes tiempos entre envíos para cada tipo de datos. Esto se debe a que, por ejemplo, un acelerómetro requiere un *data rate*⁴ rápido para poder interpretar correctamente los valores y tratar de detectar malfuncionamientos (décimas de segundo o incluso inferior), sin embargo, la temperatura por otro lado, no requiere una lectura constante de valores, si no que puede leerse con un *data rate* mucho menor ya que sería suficiente con leer el valor cada ciertos minutos por ejemplo.

Para realizar el envío se utilizará el objeto que hace referencia al servidor. Bastará con llamar a las funciones de este objeto pasando los argumentos correspondientes (identificador, tipo de datos, valores...) (Ver Ist.4.16).

Listado 4.16: Envío de datos al servidor

```
1 # Envío de datos de tres valores
2 digital_twin.putDataSetReading(self.FacilityID, self.DeviceID, ←
3     ↪ digitaltwin.TypeDataSet.Accelerometer, data)
4 # Envío de datos de un único valor
5 digital_twin.putSingleDataReading(facilityID, deviceID, ←
6     ↪ digitaltwin.TypeSingleData.Temperature, data)
```

⁴Tasa de envío de datos

4.2.4 Script de actualización de los límites

Dado que será necesario establecer los límites de los valores para cada gemelo digital se ha desarrollado un *script* para automatizar esta función.

Este *script* utiliza dos archivos de configuración que contiene toda la información necesaria. Uno contiene el identificador de la instalación y del dispositivo del gemelo digital del cual quieren establecerse los límites y la dirección del proxy. El otro contiene los valores de los límites a establecer (el archivo debe llamarse *limits.config*) (ver lst.4.17).

Listado 4.17: Archivo de configuración de los límites

```

1 [datatypes]
2 Accelerometer = true
3 Gyroscope = true
4 Magnetometer = true
5 Temperature = true
6 Pressure = true
7 Humidity = true
8 Light = true
9
10 [limits_acc] # Accelerometer
11 x_max = 1503.46
12 x_min = 556.91
13 y_max = 858.05
14 y_min = -416.97
15 z_max = 493.56
16 z_min = -494.14
17
18 [limits_gy] # Gyroscope
19 x_max = 1
20 x_min = -1
21 y_max = 1
22 y_min = -1
23 z_max = 1
24 z_min = -1
25
26 [limits_mg] # Magnetometer
27 x_max = 19.8
28 #...

```

Gracias a este *script*, para establecer el valor de los límites bastará con ejecutar el comando:

```
$ python3 ./limits_update.py --Ice.Config=gw.config
GNU/Linux
```

El criterio para definir los límites se deberá adaptar a las necesidades de cada proyecto de tal forma que se pueda detectar adecuadamente cuando se obtienen valores fuera de lo

4. RESULTADOS

normal. Una técnica que puede utilizarse consiste en medir los valores del funcionamiento normal durante un tiempo determinado. Se pueden analizar estos valores para obtener información relevante como por ejemplo la desviación estándar entre otras cosas.

Estos valores se calculan de una forma simple en la actualidad, pero se pueden cambiar por cualquier tipo de datos generados a partir de simulaciones precisas más complejas.

4.2.5 Implementación del sub-objetivo 4

El cuarto sub-objetivo consiste en el diseño y desarrollo de un entorno de pruebas sobre un motor eléctrico real que pruebe el correcto funcionamiento en conjunto de los componentes del sistema.

Para este entorno de pruebas se reutilizará un motor antiguo, el cual pertenecía a una lavadora, y se utilizará un modelo 3D que represente dicho motor eléctrico. Para el desarrollo del prototipo se utilizará:

- Un modelo 3D diseñado utilizando el software *Blender*.
- Un sensor de la compañía BOSCH así como el código que esta proporciona para el leer los datos de dicho sensor [4].
- Una *Raspberry Pi* como equipo que ejecutará el software y al cual se conectará el sensor

Es importante utilizar los mismos identificadores de la instalación y del dispositivo en ambos clientes para que el gemelo digital funcione. En este proyecto se usarán como identificadores:

- *FacilityID* → Test Facility 1
- *DeviceID* → Test Device 1

Cliente lector del sensor

En el nodo cliente que lee los datos del sensor habrá que elegir el motor a utilizar, así como el sensor que se usará para recoger datos.

En un entorno real el dispositivo a monitorear sería un motor específico o de alguna máquina de uso real en unas instalaciones. Para el entorno de pruebas se utilizará un motor que pertenecía a una lavadora (ver fig.4.21).

Como sensor se utilizará el sensor CISS de la compañía BOSCH. El sensor puede acoplarse al sensor utilizando tornillos o utilizando imanes⁵ (ver fig.4.22).

Una vez acoplado podrá medir todos los tipos de datos que se mencionan en su ficha técnica:

⁵No deben utilizarse imanes en caso de que se vaya a activar el magnetómetro ya que estos interfieren

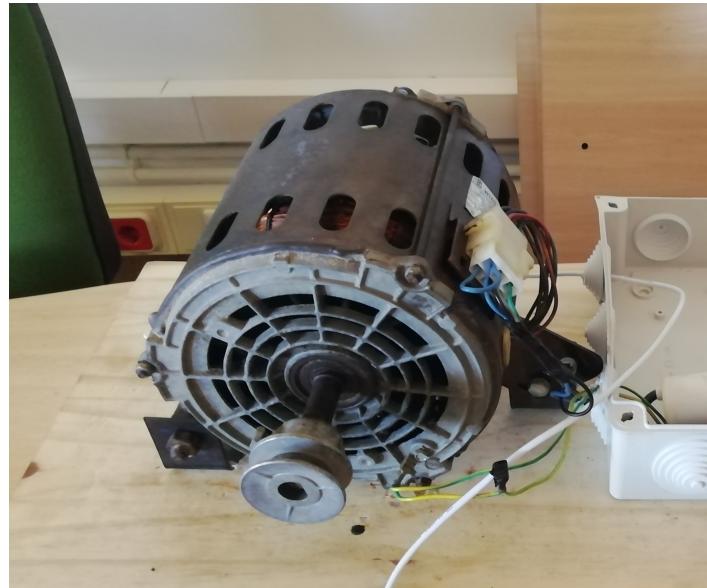


Figura 4.21: Motor eléctrico usado en el prototipo (Fotografía tomada por el autor)



Figura 4.22: Sensor acoplado al motor eléctrico (Fotografía tomada por el autor)

1. Acelerómetro
2. Giroscopio
3. Magnetómetro
4. Temperatura
5. Presión
6. Humedad
7. Luz

En la página oficial ofrecen una *demo* en lenguaje *python* que será la que se utilice en este proyecto para leer los datos. De por sí esta demo muestra la información por pantalla y la

4. RESULTADOS

guarda en archivos, por lo que simplemente hay que añadirle la parte del entorno *ZeroC Ice* explicada en la sección 4.2.3.

Además se han añadido algunos parámetros de configuración al archivo de configuración de la demo para poder elegir que datos se envían al servidor y cuales no (ver lst.4.18).

Listado 4.18: Parámetros de configuración extra

```
1 [Datatypes_configuration]
2 SendAccelerometer = true
3 SendGyroscope = true
4 SendMagnetometer = true
5 SendTemperature = true
6 SendPressure = true
7 SendHumidity = true
8 SendLight = true
```

Para simplificar las cosas simplemente se mandarán los datos al servidor en el mismo momento en que se imprimen por pantalla, teniendo en cuenta que se ha modificado el *data rate* de 100ms (10Hz) que trae configurado por defecto a 10ms (1Hz) (ver lst.4.19).

Listado 4.19: Envío de datos del sensor al servidor

```
1 # Write streaming data to the screen with 10ms = 1Hz -----
2     if ts_diff > 10:
3         if printInformation:
4             if buff[0] != '':
5                 print("TimeStamp:", tstamp, "-_Accelerometer_[mg]", "x:", ←
6                     ↪ buff[0], "y:", buff[1], "z:", buff[2])
7             if(sendAcc):
8                 data = digitaltwin.DataSet(buff[0], buff[1], buff[2])
9                 digital_twin.putDataSetReading(facilityID, deviceID, ←
10                    ↪ digitaltwin.TypeDataSet.Accelerometer, data)
11         elif buff[3] != '':
12             print("TimeStamp:", tstamp, "-_Gyroscope_[ /s ]", "x:", ←
13                 ↪ buff[3], "y:", buff[4], "z:", buff[5])
14             if(sendGy):
15                 data = digitaltwin.DataSet(buff[3], buff[4], buff[5])
16                 digital_twin.putDataSetReading(facilityID, deviceID, ←
17                    ↪ digitaltwin.TypeDataSet.Gyroscope, data)
18         elif buff[6] != '': # Magnetometro ...
19             #...
20         elif buff[9] != '':
21             print("TimeStamp:", tstamp, "-_Temperature_[ C ]:", buff[9])
22             if(sendTemp):
23                 digital_twin.putSingleDataReading(facilityID, deviceID, ←
24                    ↪ digitaltwin.TypeSingleData.Temperature, buff[9])
25             elif buff[10] != '': #Presión ...
```

```

21      ...
22      elif buff[11] != '': # Humedad ...
23          ...
24      elif buff[12] != '': # Luz ...
25          ...
26  else:
27      print("Error_in_data_string.")
28  ts_count = tstamp

```

Entorno 3D

Para preparar el entorno 3D la primera tarea puede ser establecer el modelo 3D a utilizar. Para el presente proyecto se utilizará el modelo 3D diseñado en la sección 4.1.2 (ver fig.4.23).



Figura 4.23: Modelo 3D integrado en el entorno 3D (Figura del autor)

Una vez hecho esto habrá que especificar los identificadores de la instalación y el dispositivo al que hace referencia este gemelo digital y la dirección del proxy al que se conectará (ver fig.4.24).

Establecer los límites

Para decidir el valor de los límites será necesario analizar el funcionamiento normal del motor eléctrico. Es por esto que se han medido los valores del motor eléctrico en funcionamiento normal durante unos minutos haciendo uso de la demo mencionada en la sección 4.2.5, ya que guarda un registro de los datos obtenidos.

El nivel de análisis que se realizará depende de las necesidades del proyecto que se quiera realizar, como por ejemplo la precisión con la que quiera detectarse un fallo, querer minimizar los falsos positivos de fallo, etc... En este caso, al ser el motor de una lavadora no es necesaria una precisión grande por lo que se hará un cálculo simple para obtener los valores de los límites.

4. RESULTADOS

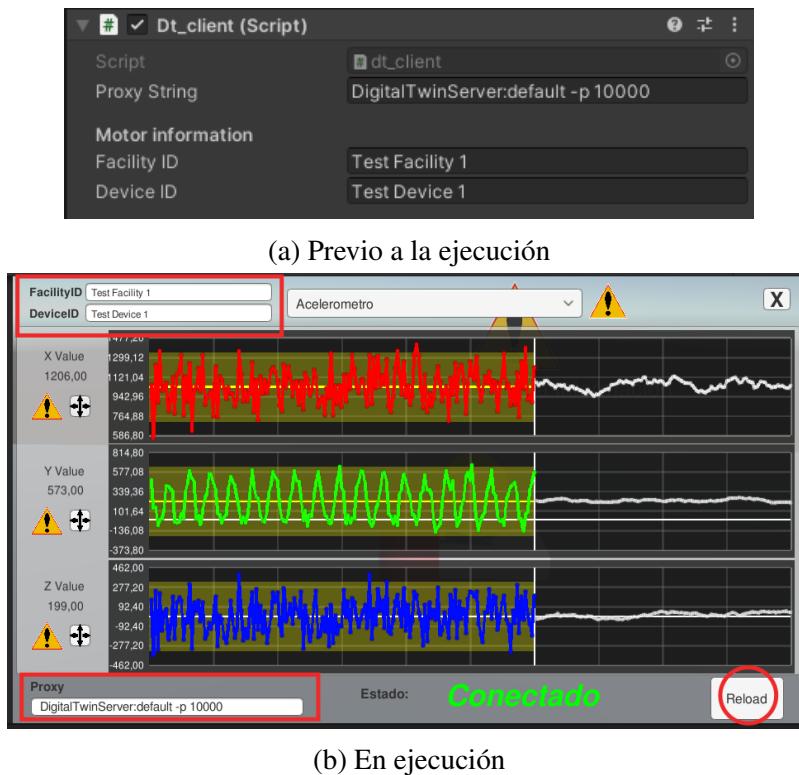


Figura 4.24: Especificación de los identificadores (Figura del autor)

Con los datos obtenidos de las medidas se ha calculado la media de los valores obtenidos así como sus desviaciones estandar (ver tab.4.2).

Para calcular los límites se utilizará el valor medio obtenido, y para calcular los límites superiores se le sumará la desviación estándar, mientras que para los límites inferiores se le restará.

Dado que la desviación estándar es la media de las desviaciones muchos valores quedarán fuera así que se cómo primera aproximación se puede probar a usar el doble de la desviación estándar a la hora de hacer los cálculos.

Sin embargo, para este proyecto el doble de la desviación estándar no es suficiente ya que los picos de valores sobrepasan los límites lo cual causaría que el sistema alertase constantemente. Es por eso que en este caso se utilizará el triple de la desviación estándar.

Un inconveniente de este procedimiento es que no es válido para valores que se espera que varíen muy lentamente o incluso no varíen, como por ejemplo la temperatura. En estos casos habría que obtener valores de intervalos de tiempo mucho más grandes, o simplemente usar como límites valores que se sabe que no deberían superarse en ningún caso de funcionamiento normal.

Para el presente proyecto los límites calculados se muestran en la tabla 4.3.

Tipo de datos	Media	Desviación estándar
<i>Acelerometro (x)</i>	1031,05	170,63
<i>Acelerometro (y)</i>	224,78	295,72
<i>Acelerometro (z)</i>	2,41	171,19
<i>Giroscopio (x)</i>	0	0
<i>Giroscopio (y)</i>	0	0
<i>Giroscopio (z)</i>	0	0
<i>Magnetometro (x)</i>	5,08	4,93
<i>Magnetometro (y)</i>	12,75	4,79
<i>Magnetometro (z)</i>	-7,22	8,06
<i>Temperatura</i>	25,7	0,01
<i>Presión</i>	943,04	0,02996
<i>Humedad</i>	26,86	0,23
<i>Luz</i>	0	0

Cuadro 4.2: Valores obtenidos del funcionamiento normal del motor eléctrico

4.3 Pruebas

A la hora de desarrollar un sistema o un software es necesario verificar el correcto funcionamiento de todas sus partes para asegurar que cumple las funciones para las que se ha desarrollado.

Existen diversas herramientas que pueden ayudar en estas funciones, pero también se pueden realizar pruebas especialmente diseñadas para el proyecto en que se está trabajando.

Para el presente proyecto es necesario verificar el funcionamiento de los tres tipos de nodos, es decir, el servidor, el cliente del entorno 3D y el cliente que lee los sensores.

4.3.1 Pruebas del servidor

El servidor contiene funciones de invocación remota por lo que cada una de las funciones que se implementen deben *testearse* convenientemente asegurándose no solo del correcto funcionamiento sino también asegurándose de que no existan vulnerabilidades que permitan acceder de forma no legítima al sistema.

Dado que en el presente proyecto se realizará un prototipo con funciones básicas estas se probarán mediante pruebas de caja negra utilizando la herramienta *PyUnit* tal y como se mencionó en el capítulo 3:Metodología.

Nótese que entre las funciones básicas hay ciertas funciones (*put* y *update*) con las que el servidor guardará los datos recibidos en la base de datos y con otras funciones (*get*) devolverá el último valor guardado. Teniendo esto en cuenta, una forma de realizar pruebas unitarias pero a la vez probando el funcionamiento completo consistiría en los dos pasos (ver lst.4.20).

Primero se enviará un número aleatorio para que el servidor lo guarde en la base de datos,

4. RESULTADOS

Tipo de datos	Límites eje x	Límites eje y	Límites eje z
<i>Acelerometro</i>	Superior: 1542,94	Superior: 1111,92	Superior: 515,98
	Inferior: 519,16	Inferior: -662,37	Inferior: -511,16
<i>Giroscopio</i>	Superior: 1	Superior: 1	Superior: 1
	Inferior: -1	Inferior: -1	Inferior: -1
<i>Magnetometro</i>	Superior: 19,87	Superior: 27,14	Superior: 16,95
	Inferior: -9,7	Inferior: -1,63	Inferior: -31,4
<i>Temperatura</i>	Superior: 34	Superior: -	Superior: -
	Inferior: 20	Inferior: -	Inferior: -
<i>Presión</i>	Superior: 943,13	Superior: -	Superior: -
	Inferior: 942,95	Inferior: -	Inferior: -
<i>Humedad</i>	Superior: 27,57	Superior: -	Superior: -
	Inferior: 26,16	Inferior: -	Inferior: -
<i>Luz</i>	Superior: 1	Superior: -	Superior: -
	Inferior: -1	Inferior: -	Inferior: -

Cuadro 4.3: Límites establecidos para detección del correcto funcionamiento

y si no hay ningún error, la prueba se completará correctamente.

En el segundo paso se solicitará el último valor con las funciones *get* y en este caso se comprobará que el número recibido coincida con aquel que se envió en el paso anterior, si esto se cumple y no hay ningún error la prueba se completará correctamente.

Listado 4.20: Pruebas unitarias al servidor

```

1  class PruebaUnitaria(unittest.TestCase):
2      # Put (Paso 1)
3      def test_putSingleData(self):
4          dt.putSingleDataReading(FACILITY_ID, DEVICE_ID, ←
5              ↪ digitaltwin.TypeSingleData.Temperature, num)
6
7      def test_putDataSet(self):
8          dt.putDataSetReading(FACILITY_ID, DEVICE_ID, ←
9              ↪ digitaltwin.TypeDataSet.Accelerometer, digitaltwin.DataSet(num, ←
10                 ↪ num, num))
11
12
13     # Get (Paso 2)
14     def test_getSingleData(self):
15         a = dt.getSingleDataReading(FACILITY_ID, DEVICE_ID, ←
16             ↪ digitaltwin.TypeSingleData.Temperature)
17         self.assertEqual(round(a, 2), num)
18
19
20     def test_getDataSet(self):
21         b = dt.getDataSetReading(FACILITY_ID, DEVICE_ID, ←
22             ↪ digitaltwin.TypeDataSet.Accelerometer)
23         self.assertEqual(round(b.x, 2), num)
24
25     # Update limits (Paso 1)
26     def test_updateSingleDataLimits(self): ...

```

```

18
19     def test_updateDataSetLimits(self): ...
20 # Get limits (Paso 2)
21     def test_getSingleDataLimits(self): ...
22
23     def test_getDataSetLimits(self): ...

```

4.3.2 Pruebas del cliente del entorno 3D

El gemelo digital del entorno 3D está formado por varios componentes que habrá que verificar de manera individual con pruebas unitarias, como, por ejemplo, el componente que muestra una gráfica. En todo software habrá que realizar este tipo de pruebas al implementar nuevas funciones ya que si no se hiciese podría aparecer un error en el futuro que sería mucho más difícil de identificar.

Ahora, la parte más importante del entorno 3D es el funcionamiento del gemelo digital completo para que cumpla todas las funciones. El funcionamiento del gemelo digital es definido por los valores que recibe del servidor y que corresponden a un motor real. Por ello, la mejor opción podría ser hacer uso de un motor real que envíe los datos al servidor de forma que se pueda comprobar el funcionamiento con valores reales. No obstante, esta opción no permitiría verificar el funcionamiento en situaciones en que se den datos erróneos, situaciones las cuales también deben ser *testeadas*.

Teniendo esto en cuenta se implementará un pequeño programa que generé datos según unos parámetros para permitir a los usuarios probar el entorno 3D ante valores específicos.

Archivo de configuración

Para esto se hará uso de un archivo de configuración que contendrá los identificadores, el proxy al que conectarse, el *data rate* y dos valores para cada eje de cada tipo de datos (ver lst.4.21).

Listado 4.21: Configuración del nodo cliente de pruebas

```

1 [zeroc_ice]
2 dt.Server=DigitalTwinServer:default -p 10000
3
4 FacilityID=Simulation_Facility1
5 DeviceID=Simulation_Device1
6
7 send_rate=0.01
8
9 # Accelerometer
10 [limits_acc]
11 x_max = 1503.46

```

4. RESULTADOS

```
12 x_min = 556.91
13 y_max = 858.05
14 y_min = -416.97
15 z_max = 493.56
16 z_min = -494.14
17
18 # Gyroscope
19 [limits_gy]
20 ...
```

Valores generados

El programa generará datos oscilando entre esos dos valores y los mandará según el *data rate* especificado (ver fig.4.25).

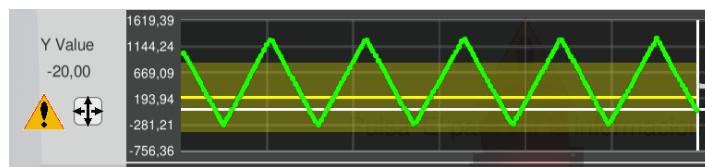


Figura 4.25: Valores generados con cliente de pruebas (Figura del autor)

4.3.3 Pruebas del cliente lector de sensores

Cabe recordar que en este nodo se ha utilizado un código proporcionado por la empresa BOSCH que será el que leerá los sensores.

Dado que solo es necesario añadir la parte relacionada con *ZeroC Ice* para realizar el envío no es necesario hacer muchas pruebas, ya que el código está convenientemente *testeado* por la propia empresa.

4.4 Costes, planificación y presupuesto

En esta sección se explicarán los costes a tener en cuenta para el cálculo del coste total del prototipo considerando tanto el hardware como los recursos humanos necesarios para su desarrollo.

El coste del hardware variará, obviamente, según los componentes elegidos para el prototipo. Para este caso se tendrá en cuenta que el cliente de representación 3D y el contenedor *docker* se ejecutan en la misma máquina, pero en caso de que el servidor se ejecutase en otra máquina o se hiciese uso de un servicio en la nube habría que añadir estos costes.

El listado de precios de los componentes utilizados en el prototipo explicado durante el presente trabajo pueden verse en la tabla 4.4.

A la hora de calcular el coste de los recursos humanos será necesario tener en cuenta las horas de trabajo necesarias para realizar todas las tareas del proceso.

Componente	Precio
<i>Ordenador portátil (Lenovo IdeaPad5)</i>	898,00€
<i>Raspberry Pi 4</i>	194,99€
<i>Sensor (BOSCH CISS)</i>	50€
<i>Motor eléctrico reutilizado</i>	30€
TOTAL	1172,99€

Cuadro 4.4: Costes de los recursos hardware utilizados en el prototipo

En la tabla 4.5 puede verse un resumen de las horas dedicadas a diferentes tareas durante el presente proyecto así como el coste de cada tarea suponiendo un salario de 35€ la hora.

Tarea	Horas	Coste
<i>Diseño de la arquitectura</i>	10	350€
<i>Diseño de la base de datos</i>	6	210€
<i>Diseño de la interfaz ZeroC Ice</i>	10	350€
<i>Diseño del modelo 3D</i>	10	350€
<i>Implementación del servidor</i>	40	1400€
<i>Implementación del cliente de representación 3D</i>	120	4200€
<i>Implementación del cliente lector de sensores</i>	16	560€
<i>Preparación del motor+sensor+raspberry</i>	24	840€
<i>Pruebas</i>	6	210€
TOTAL	242	8470€

Cuadro 4.5: Costes de los recursos humanos empleados durante el desarrollo del prototipo

Coste total

Tras haber desglosado los costes del hardware y los recursos humanos el coste total del prototipo del presente proyecto es de 9642,99€.

Capítulo 5

Conclusiones

En este capítulo se realizará un juicio crítico sobre los resultados obtenidos en el presente proyecto repasando los objetivos originales y lo que se ha generado a partir de cada uno de estos.

El objetivo principal del proyecto consistía en implementar una arquitectura de gemelo digital para motores eléctricos con un entorno de pruebas que utilice un pequeño prototipo. Para conseguir este objetivo se dividió en varios sub-objetivos.

Para el desarrollo del presente proyecto se ha seguido un proceso de desarrollo siguiendo la metodología Rapid Application Developmnent (RAD), de tal forma que se han desarrollado en paralelo los diferentes sub-objetivos permitiendo adaptar los diseños e implementaciones según apareciesen nuevas necesidades y/o modificaciones.

En primer lugar, se diseñó una arquitectura que permitiese comunicar los tres tipos de nodos y una base de datos para almacenar valores. Estos diseños posteriormente se mejoran para permitir que el sistema sea escalable y fácilmente adaptable a posibles nuevas necesidades.

Se ha desarrollado también una interfaz haciendo uso del middleware *ZeroC Ice* que será la que usarán los nodos para comunicarse e intercambiar datos mediante invocaciones remotas.

El *primer sub-objetivo* consistía en desarrollar un software que recopile y almacene los datos y el estado de determinados recursos físicos. Para cumplir este sub-objetivo se ha desarrollado un servidor que implementa la interfaz *Ice* de tal forma que los clientes puedan realizar invocaciones remotas a sus funciones. También almacena los *thresholds* o límites para los diferentes valores de cada gemelo digital, de tal forma que se pueda diferenciar el funcionamiento normal del erróneo. También se ha hecho uso de la herramienta *Docker* para construir un contenedor listo para ejecutarse en diferentes máquinas.

El *segundo sub-objetivo* consistía en desarrollar un software que muestre una interfaz de visualización 3D con capacidad de predicción/simulación y representación de la información asociada a un recurso físico. Para cumplir esta función se ha hecho uso de la herramienta *Unity* para crear un entorno 3D en el que se han desarrollado los componentes necesarios:

5. CONCLUSIONES

componente (cliente) que se conecta al servidor y solicita los datos, componentes gráficos que muestra los datos, pequeña predicción de datos futuros y otros componentes auxiliares.

El *tercer sub-objetivo* consistía en desarrollar un software que obtenga datos de los recursos físicos. Para este sub-objetivo se ha desarrollado un software que obtiene los datos de un sensor conectado a un recurso real para enviarlos al servidor cumpliendo así la función de cliente.

El *cuarto sub-objetivo* consistía en desarrollar un entorno de pruebas para probar los diferentes componentes del sistema. Para ello se ha hecho uso de un motor eléctrico viejo que pertenecía a una lavadora, un sensor CISS de la empresa *BOSCH* así como la demo que esta proporciona, y una Raspberry Pi para obtener los datos del sensor y enviarlos al servidor. También se han tomado medidas de los valores del motor en funcionamiento para poder establecer unos *thresholds* o límites adecuados.

También se han implementado algunas pruebas básicas para verificar los funcionamientos de las diferentes partes.

Como resultado de estas implementaciones se ha obtenido un prototipo de un gemelo digital de un motor eléctrico y el diseño para un sistema más amplio que haga uso de varios gemelos digitales.

5.1 Trabajos futuros

En esta sección se mencionarán algunos trabajos que podrían realizarse en el futuro a partir del trabajo realizado en el presente proyecto.

En primer lugar, podría implementarse un sistema que incluyese actuadores además de sensores de tal forma que se pueda automatizar más el funcionamiento del sistema para evitar problemas.

Otra posibilidad consistiría en implementar un sistema de varias máquinas que realmente interactúen directamente entre ellas siendo este un entorno en el que el fallo de una afecta al funcionamiento del sistema completo, y por tanto, resultarían especialmente útiles los actuadores mencionados anteriormente.

Por otro lado, existe la posibilidad de realizar unas simulaciones adaptadas a un sistema concreto según sus necesidades, ya sea para mantenimiento predictivo, lo cual evitaría fallos, o para simulación de diseños, lo cual disminuiría los costes de añadir un nuevo elemento.

También puede implementarse un sistema de alertas adecuado que active alarmas si fuese necesario, aumentando así el nivel de seguridad de las instalaciones y sus empleados.

5.2 Justificación de competencias adquiridas

En esta sección se explicarán la aplicación de las competencias específicas adquiridas en la tecnología cursada.

En el TFG se han aplicado las competencias correspondientes a la Tecnología Específica de *ingeniería de computadores*:

IC3: *[Capacidad de analizar y evaluar arquitecturas de computadores, incluyendo plataformas paralelas y distribuidas, así como desarrollar y optimizar software para las mismas.]*

Para lograr el desarrollo de un gemelo digital se ha diseñado y desarrollado tanto la arquitectura del sistema distribuido consiguiendo escalabilidad como el software necesario en los diferentes nodos del sistema.

IC4: *[Capacidad de diseñar e implementar software de sistema y de comunicaciones.]*

En el presente TFG se ha diseñado una interfaz *Ice* que es la que se utiliza para comunicar los diferentes nodos del sistema mediante invocaciones remotas.

ANEXOS

Anexo A

Repositorio

El resultado del desarrollo del presente trabajo puede encontrarse en el repositorio *Bitbucket* dedicado a ello:

<https://bitbucket.org/danielej/tfg.danielespinar/src/master/> [6]

Directorio *src*

La carpeta *src/dt/* contiene el código fuente, archivos de configuración, *Dockerfiles* correspondientes a la implementación del nodo servidor (sub-objetivo 1).

La carpeta *src/monitor/* contiene el código fuente y archivos de configuración correspondientes a la implementación del nodo cliente encargado de la representación del entorno 3D (sub-objetivo 2).

La carpeta *src/gw/* contiene el código fuente y archivos de configuración correspondientes a la implementación del nodo cliente encargado de leer valores del sensor (sub-objetivo 3).

La carpeta *src/limit_manager/* contiene el código fuente y archivos de configuración correspondientes a la implementación del programa usado para establecer límites de los valores del gemelo digital.

La carpeta *src/testing/* contiene el código fuente y archivos de configuración correspondientes a las pruebas unitarias del servidor (*src/testing/server_tests.py*) y las pruebas del entorno 3D (*src/testing/gw_sim.py*).

La carpeta *src/slice/* contiene el archivo que define la interfaz *Ice* que se utilizará en el sistema.

Referencias

- [1] Ben Lutkevich. Metodología de análisis y diseño de sistemas estructurado SSADM. URL: <https://www.techtarget.com/searchsoftwarequality/definition/SSADM>, Junio 2022. Último acceso: jun. 2022.
- [2] Bitbucket. Página web de bitbucket. URL: <https://bitbucket.org/>, Junio 2022. Último acceso: jun. 2022.
- [3] Blender. Página web de blender. URL: <https://www.blender.org/>, Junio 2022. Último acceso: jun. 2022.
- [4] BOSCH. Página web con toda la información relativa al sensor. URL: <https://www.bosch-connectivity.com/products/industry-4-0/connected-industrial-sensor-solution/downloads/>, Junio 2022. Último acceso: jun. 2022.
- [5] Code Monkey. Tutoríal de creación de un gráfico en unity. URL: <https://www.youtube.com/playlist?list=PLzDRvYVwl53v5ur4GluobyckImZz3TVQ>, Abril 2019. Último acceso: jun. 2022.
- [6] Daniel Espinar Jiménez. Repositorio *Bitbucket* de este tfg. URL: <https://bitbucket.org/danielej/tfg.danielespinar/src/master/>, Julio 2022. Último acceso: jul. 2022.
- [7] Docker. Página web de docker. URL: <https://www.docker.com/>, Junio 2022. Último acceso: jun. 2022.
- [8] Escuela Superior de Informática, Universidad de Castilla-La Mancha. Guía de estilo y formato para Trabajos Fin de Grado. URL: <https://pruebasaluuclm.sharepoint.com/sites/esicr/tfg/SiteAssets/SitePages/Inicio/20190304-GuiaEstiloFormatoTFG.pdf>, Marzo 2019. Último acceso: jun. 2022.
- [9] Goran Jevtic. Ciclo de vida del desarrollo del software *Software Development Life-Cycle*. URL: <https://phoenixnap.com/blog/software-development-life-cycle#:~:text=Software%20Development%20Life%20Cycle%20is,%2C%20Test%2C%20Deploy%2C%20Maintain.>, Mayo 2019. Último acceso: jun. 2022.

- [10] Kissflow. Desarrollo rápido de aplicaciones RAD. URL: <https://kissflow.com/low-code/rad/rapid-application-development/>, Septiembre 2021. Último acceso: jun. 2022.
- [11] LibreOffice. Página web de libreoffice calc. URL: <https://es.libreoffice.org/descubre/calc/>, Junio 2022. Último acceso: jun. 2022.
- [12] MathWorks. Explicación del gemelo digital. URL: <https://explore.mathworks.com/digital-twins-for-predictive-maintenance>, Junio 2022. Último acceso: jun. 2022.
- [13] Metodologías de Software. Modelo en espiral de barry boehm. URL: <https://metodologiassoftware.wordpress.com/2017/11/29/modelo-en-espiral/>, Noviembre 2017. Último acceso: jun. 2022.
- [14] Microsoft. Definición de middleware. URL: <https://azure.microsoft.com/es-es/overview/what-is-middleware/>, Junio 2022. Último acceso: jun. 2022.
- [15] Monstruos del diseño. Tipos y técnicas de modelado 3d. URL: <https://monstruosdeldiseno.com/podcast/tipos-tecnicas-modelado-3d>, Junio 2022. Último acceso: jun. 2022.
- [16] Patrick McCarthy. Repositorio del paquete NuGet. URL: <https://github.com/GlitchEnzo/NuGetForUnity>, Abril 2022. Último acceso: jun. 2022.
- [17] PostgreSQL. Página de web de psotgresql. URL: <https://www.postgresql.org/>, Junio 2022. Último acceso: jun. 2022.
- [18] ProfessionalQA. Metodología de diseño orientado a objetos OOD. URL: <https://www.professionalqa.com/object-oriented-design>, Junio 2018. Último acceso: jun. 2022.
- [19] Python. Documentación del paquete configparser. URL: <https://docs.python.org/3/library/configparser.html>, Junio 2022. Último acceso: jun. 2022.
- [20] Python. Página de explicación de pyunit. URL: <https://wiki.python.org/moin/PyUnit>, Junio 2022. Último acceso: jun. 2022.
- [21] Python. Página web de python. URL: <https://www.python.org/>, Junio 2022. Último acceso: jun. 2022.
- [22] Red Hat. Metodología ágil. URL: <https://www.redhat.com/es/devops/what-is-agile-methodology>, Enero 2020. Último acceso: jun. 2022.
- [23] SQLAlchemy. Página web de sqlalchemy. URL: <https://www.sqlalchemy.org/>, Junio 2022. Último acceso: jun. 2022.
- [24] Unity.

- [25] Unity. Página web de unity. URL: <https://unity.com/es>, Junio 2022. Último acceso: jun. 2022.
- [26] Unity. Tienda de *assets* de unity. URL: <https://assetstore.unity.com/>, Junio 2022. Último acceso: jun. 2022.
- [27] Visual Studio. Descargar visual studio code. URL: <https://code.visualstudio.com/>, Junio 2022. Último acceso: jun. 2022.
- [28] Visual Studio Code. Uso de visual studio code para desarrollo unity. URL: <https://code.visualstudio.com/docs/other/unity>, Mayo 2022. Último acceso: jun. 2022.
- [29] ZeroC. Página de web del middleware zeroc ice. URL: <https://zeroc.com/products/ice>, Junio 2022. Último acceso: jun. 2022.

Este documento fue editado y tipografiado con L^AT_EX empleando
la clase **esi-tfg** (versión 0.20181017) que se puede encontrar en:
<https://github.com/UCLM-ESI/esi-tfg>

[]

